南京航空航天大学数据结构课设(2024)

姓名:邓琳超
班级:1623302
学号:162330222
指导老师:高航

目录

必做题目

选做题目

总结

> 1. **反思**

必做题目介绍

1.系统进程统计

题目:

设计一个程序，每秒统计一次当前系统的进程状况，
并按照内存使用自多到少排序打印输出相关信息。
对已经结束的进程，另外给出一个列表，并显示该进程的结束时间和持续时间。

数据结构:

建立结构体**ProcessInfo**来存储进程信息,再建立两个链表**ActiveHead**和**EndedHead**来存储活动进程和已结束进程.

```
struct ProcessInfo {
    int processID;              // 进程ID
    string processName;         // 进程名称
    long long memoryUsage;      // 进程内存使用量（以字节为单位）
    int duration;               // 进程的持续时间（秒）
};

struct ActiveProcessNode {
    ProcessInfo data;          // 存储进程信息的结构体
    ActiveProcessNode* next;   // 指向下一个活动进程节点的指针
};

struct EndedProcessNode {
    ProcessInfo data;          // 存储进程信息的结构体
    EndedProcessNode* prev;    // 指向前一个已结束进程节点的指针
    EndedProcessNode* next;    // 指向下一个已结束进程节点的指针
};
```

算法思想:

使用**ps**的库来获取进程信息,并根据时间以及内存大小来进行排序.
注意:建议使用msvc编译器,不然会很麻烦.

测试结果(部分):

```
Ended Processes:
Process ID: 22584, Name: chrome.exe, Duration: 3s
Process ID: 30060, Name: chrome.exe, Duration: 3s
Process ID: 38700, Name: BackgroundDownload.exe, Duration: 8s
Active Processes:
Process ID: 15272, Name: Cursor.exe, Memory Usage: 683913216 KB, Duration: 12s
Process ID: 36732, Name: devenv.exe, Memory Usage: 682258432 KB, Duration: 12s
Process ID: 36028, Name: Cursor.exe, Memory Usage: 509796352 KB, Duration: 12s
Process ID: 31860, Name: chrome.exe, Memory Usage: 403763200 KB, Duration: 12s
Process ID: 36884, Name: Cursor.exe, Memory Usage: 367247360 KB, Duration: 12s
Process ID: 34616, Name: chrome.exe, Memory Usage: 358215680 KB, Duration: 12s
Process ID: 12356, Name: SearchHost.exe, Memory Usage: 341561344 KB, Duration: 12s
Process ID: 21460, Name: QQ.exe, Memory Usage: 337285120 KB, Duration: 12s
Process ID: 5304, Name: chrome.exe, Memory Usage: 294674432 KB, Duration: 12s
Process ID: 14416, Name: AppleMusic.exe, Memory Usage: 291160064 KB, Duration: 12s
Process ID: 14700, Name: Explorer.EXE, Memory Usage: 280494080 KB, Duration: 12s
Process ID: 8820, Name: chrome.exe, Memory Usage: 266489856 KB, Duration: 12s
Process ID: 16760, Name: WINWORD.EXE, Memory Usage: 258449408 KB, Duration: 12s
Process ID: 29032, Name: chrome.exe, Memory Usage: 256167936 KB, Duration: 12s
Process ID: 24252, Name: QQ.exe, Memory Usage: 240713728 KB, Duration: 12s
Process ID: 10308, Name: chrome.exe, Memory Usage: 237404160 KB, Duration: 12s
Process ID: 25568, Name: steamwebhelper.exe, Memory Usage: 214949888 KB, Duration: 12s
Process ID: 19668, Name: chrome.exe, Memory Usage: 202682368 KB, Duration: 12s
Process ID: 15308, Name: msedge.exe, Memory Usage: 201084928 KB, Duration: 12s
Process ID: 21792, Name: msedge.exe, Memory Usage: 192311296 KB, Duration: 12s
Process ID: 12832, Name: chrome.exe, Memory Usage: 164335616 KB, Duration: 12s
Process ID: 37320, Name: chrome.exe, Memory Usage: 160530432 KB, Duration: 12s
Process ID: 22560, Name: PhoneExperienceHost.exe, Memory Usage: 156217344 KB, Duration: 12s
Process ID: 19684, Name: Lingma.exe, Memory Usage: 155942912 KB, Duration: 12s
Process ID: 25492, Name: ServiceHub.ThreadedWaitDialog.exe, Memory Usage: 155508736 KB, Duratio
Process ID: 17560, Name: chrome.exe, Memory Usage: 154484736 KB, Duration: 12s
Process ID: 6260, Name: cpptools-srv.exe, Memory Usage: 154062848 KB, Duration: 12s
Process ID: 37840, Name: cpptools-srv.exe, Memory Usage: 153202688 KB, Duration: 12s
Process ID: 32884, Name: copilot-language-server.exe, Memory Usage: 151412736 KB, Duration: 8s
Process ID: 19064, Name: TextInputHost.exe, Memory Usage: 150327296 KB, Duration: 12s
Process ID: 3236, Name: steamwebhelper.exe, Memory Usage: 147841024 KB, Duration: 12s
Process ID: 24388, Name: Cursor.exe, Memory Usage: 146423808 KB, Duration: 12s
Process ID: 8884, Name: Cursor.exe, Memory Usage: 144760832 KB, Duration: 12s
Process ID: 19028, Name: AMPLibraryAgent.exe, Memory Usage: 143028224 KB, Duration: 12s
Process ID: 25300, Name: cpptools-srv.exe, Memory Usage: 140148736 KB, Duration: 12s
Process ID: 29160, Name: cpptools-srv.exe, Memory Usage: 139943936 KB, Duration: 12s
```

```
Process ID: 9664, Name: ServiceHub.IdentityHost.exe, Memory Usage: 132743168 KB, Duration: 12s
Process ID: 19832, Name: chrome.exe, Memory Usage: 132108288 KB, Duration: 12s
Process ID: 6960, Name: ServiceHub.VSDetouredHost.exe, Memory Usage: 131043328 KB, Duration: 12s
Process ID: 32960, Name: ServiceHub.Host.dotnet.x64.exe, Memory Usage: 128376832 KB, Duration: 8
Process ID: 14104, Name: OneDrive.exe, Memory Usage: 126533632 KB, Duration: 12s
Process ID: 19320, Name: cpptools-srv.exe, Memory Usage: 125394944 KB, Duration: 12s
```

源代码：

```cpp
#include <iostream>
#include <windows.h>
#include <psapi.h>
#include <string>
#include <chrono>
#include <thread>
using namespace std;

struct ProcessInfo {
    int processID;              // 进程ID
    string processName;         // 进程名称
    long long memoryUsage;      // 进程内存使用量（以字节为单位）
    int duration;               // 进程的持续时间（秒）
};

struct ActiveProcessNode {
    ProcessInfo data;           // 存储进程信息的结构体
    ActiveProcessNode* next;    // 指向下一个活动进程节点的指针
};

struct EndedProcessNode {
    ProcessInfo data;           // 存储进程信息的结构体
    EndedProcessNode* prev;     // 指向前一个已结束进程节点的指针
    EndedProcessNode* next;     // 指向下一个已结束进程节点的指针
};

ActiveProcessNode* activeHead = nullptr;    // 活动进程链表的头指针
EndedProcessNode* endedHead = nullptr;     // 已结束进程链表的头指针

// 向活动进程链表中插入新进程
void InsertActiveProcess(ProcessInfo process) {
    ActiveProcessNode* newNode = new ActiveProcessNode{ process, nullptr }; // 创建新的活动进程节
    if (!activeHead || activeHead->data.memoryUsage < process.memoryUsage) {
        newNode->next = activeHead; // 如果链表为空或当前进程内存使用大于头节点，则插入到链表头
        activeHead = newNode;
    }
    else {
        ActiveProcessNode* current = activeHead;
        while (current->next && current->next->data.memoryUsage >= process.memoryUsage) {
            current = current->next; // 按照内存使用量排序，找到合适的位置插入
        }
```

```
        newNode->next = current->next;  // 插入新节点
        current->next = newNode;
    }
}

// 向已结束进程链表中插入新进程
void InsertEndedProcess(ProcessInfo process) {
    EndedProcessNode* newNode = new EndedProcessNode{ process, nullptr, nullptr }; // 创建新的已
    if (!endedHead || endedHead->data.duration > process.duration) {
        newNode->next = endedHead;   // 如果链表为空或当前进程持续时间小于头节点，则插入到链表头
        if (endedHead) endedHead->prev = newNode;
        endedHead = newNode;
    }
    else {
        EndedProcessNode* current = endedHead;
        while (current->next && current->next->data.duration <= process.duration) {
            current = current->next;   // 按照持续时间排序，找到合适的位置插入
        }
        newNode->next = current->next; // 插入新节点
        if (current->next) current->next->prev = newNode;
        current->next = newNode;
        newNode->prev = current;
    }
}

// 更新活动进程的持续时间，每次调用时持续时间加 1
void UpdateProcessDuration() {
    ActiveProcessNode* current = activeHead;
    while (current) {
        current->data.duration++;  // 每秒钟持续时间加 1
        current = current->next;
    }
}

// 检查已结束进程是否重新启动
void CheckForRestartedProcesses() {
    EndedProcessNode* current = endedHead;
    while (current) {
        // 检查该进程是否已经重新出现在活动链表中
        ActiveProcessNode* checkNode = activeHead;
        bool found = false;
        while (checkNode) {
            if (checkNode->data.processID == current->data.processID) {
```

```
                found = true;  // 如果找到了重新启动的进程
                break;
            }
            checkNode = checkNode->next;
        }

        // 如果进程已重新启动，从已结束链表中移除它
        if (found) {
            // 删除该节点
            if (current->prev) {
                current->prev->next = current->next;
            }
            else {
                endedHead = current->next;
            }
            if (current->next) {
                current->next->prev = current->prev;
            }
            EndedProcessNode* temp = current;
            current = current->next;
            delete temp;  // 删除已结束进程节点
        }
        else {
            current = current->next;
        }
    }
}

// 更新活动进程链表
void UpdateProcesses() {
    int processes[1024], needed, count;
    EnumProcesses((DWORD*)processes, sizeof(processes), (DWORD*)&needed); // 获取当前系统中的所有
    count = needed / sizeof(int);  // 计算实际的进程数

    // 更新活动进程的持续时间
    UpdateProcessDuration();

    // 遍历系统中的每个进程
    for (int i = 0; i < count; i++) {
        if (processes[i] != 0) {
            HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE, pı
            if (hProcess) {
                PROCESS_MEMORY_COUNTERS pmc;
```

```
            if (GetProcessMemoryInfo(hProcess, &pmc, sizeof(pmc))) {
                string processName = "<unknown>";  // 使用标准字符串类型来存储进程名称
                HMODULE hMod;
                DWORD cbNeeded;
                if (EnumProcessModules(hProcess, &hMod, sizeof(hMod), &cbNeeded)) {
                    char name[MAX_PATH];
                    GetModuleBaseNameA(hProcess, hMod, name, sizeof(name)); // 获取进程名称
                    processName = name;  // 转换为 C++ string
                }
                ProcessInfo processInfo = { processes[i], processName, pmc.WorkingSetSize,

                // 判断该进程是否在活动链表中，如果没有就插入
                bool found = false;
                ActiveProcessNode* checkNode = activeHead;
                while (checkNode) {
                    if (checkNode->data.processID == processes[i]) {
                        found = true;  // 如果进程已经在链表中
                        break;
                    }
                    checkNode = checkNode->next;
                }
                if (!found) {
                    InsertActiveProcess(processInfo); // 如果没有找到该进程，插入到活动链表中
                }
            }
            CloseHandle(hProcess);
        }
    }
}

// 遍历活动进程链表，将已经结束的进程移到已结束链表
ActiveProcessNode* prev = nullptr;
ActiveProcessNode* current = activeHead;
while (current) {
    HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, current->data.processID
    if (!hProcess) {
        // 进程结束，将其移到已结束链表
        ProcessInfo endedProcess = current->data;
        InsertEndedProcess(endedProcess); // 插入到已结束链表

        // 删除当前节点
        if (prev) {
            prev->next = current->next;
```

```cpp
            }
            else {
                activeHead = current->next;
            }
            ActiveProcessNode* temp = current;
            current = current->next;
            delete temp;  // 删除当前活动进程节点
        }
        else {
            prev = current;
            current = current->next;
            CloseHandle(hProcess);
        }
    }

    // 检查已结束进程是否重新启动
    CheckForRestartedProcesses();
}

// 打印活动进程信息
void PrintActiveProcesses() {
    cout << "Active Processes:" << endl;
    ActiveProcessNode* current = activeHead;
    while (current) {
        cout << "Process ID: " << current->data.processID << ", Name: " << current->data.proces
            << ", Memory Usage: " << current->data.memoryUsage << " KB, Duration: " << current-
        current = current->next;
    }
}

// 打印已结束进程信息
void PrintEndedProcesses() {
    cout << "Ended Processes:" << endl;
    EndedProcessNode* current = endedHead;
    while (current) {
        cout << "Process ID: " << current->data.processID << ", Name: " << current->data.proces
            << ", Duration: " << current->data.duration << "s" << endl;
        current = current->next;
    }
}

int main() {
    while (true) {
```

```
        UpdateProcesses();           // 更新进程信息
        PrintActiveProcesses();      // 打印当前活动进程
        PrintEndedProcesses();       // 打印已结束进程
        this_thread::sleep_for(chrono::seconds(1));  // 每秒钟刷新一次
    }
    return 0;
}
```

2.栈与队列

题目:

八皇后问题.

数据结构:

使用来自己实现**栈**的模板模拟dfs的递归与回溯,再通过自己实现**pair**来存储坐标.

```cpp
template <typename T1, typename T2>
struct MyPair {
    T1 first;
    T2 second;

    MyPair() : first(), second() {}
    MyPair(T1 a, T2 b) : first(a), second(b) {}
};

// 自定义栈结构体
template <typename T>
struct MyStack {
    struct Node {
        T data;
        Node* next;
        Node(T val) : data(val), next(nullptr) {}
    };

    Node* topNode;  // 栈顶元素

    MyStack() : topNode(nullptr) {}

    // 判断栈是否为空
    bool empty();

    // 入栈
    void push(const T& value);

    // 出栈
    void pop();

    // 获取栈顶元素
    T top();

    // 析构栈，释放内存
    ~MyStack() ;
};
```

算法思想:

建立数组存储对角线,行和列来标记是否有**皇后**,然后利用栈的进出模拟递归回溯**dfs**.

测试结果：

```
1 5 8 6 3 7 2 4
1 6 8 3 7 4 2 5
1 7 4 6 8 2 5 3
1 7 5 8 2 4 6 3
2 4 6 8 3 1 7 5
2 5 7 1 3 8 6 4
2 5 7 4 1 8 6 3
2 6 1 7 4 8 3 5
2 6 8 3 1 4 7 5
2 7 3 6 8 5 1 4
2 7 5 8 1 4 6 3
2 8 6 1 3 5 7 4
3 1 7 5 8 2 4 6
3 5 2 8 1 7 4 6
3 5 2 8 6 4 7 1
3 5 7 1 4 2 8 6
3 5 8 4 1 7 2 6
3 6 2 5 8 1 7 4
3 6 2 7 1 4 8 5
3 6 2 7 5 1 8 4
3 6 4 1 8 5 7 2
3 6 4 2 8 5 7 1
3 6 8 1 4 7 5 2
3 6 8 1 5 7 2 4
3 6 8 2 4 1 7 5
3 7 2 8 5 1 4 6
3 7 2 8 6 4 1 5
3 8 4 7 1 6 2 5
4 1 5 8 2 7 3 6
4 1 5 8 6 3 7 2
4 2 5 8 6 1 3 7
4 2 7 3 6 8 1 5
4 2 7 3 6 8 5 1
4 2 7 5 1 8 6 3
4 2 8 5 7 1 3 6
4 2 8 6 1 3 5 7
4 6 1 5 2 8 3 7
4 6 8 2 7 1 3 5
4 6 8 3 1 7 5 2
4 7 1 8 5 2 6 3
4 7 3 8 2 5 1 6
```

4 7 5 2 6 1 3 8

4 7 5 3 1 6 8 2

4 8 1 3 6 2 7 5

4 8 1 5 7 2 6 3

4 8 5 3 1 7 2 6

5 1 4 6 8 2 7 3

5 1 8 4 2 7 3 6

5 1 8 6 3 7 2 4

5 2 4 6 8 3 1 7

5 2 4 7 3 8 6 1

5 2 6 1 7 4 8 3

5 2 8 1 4 7 3 6

5 3 1 6 8 2 4 7

5 3 1 7 2 8 6 4

5 3 8 4 7 1 6 2

5 7 1 3 8 6 4 2

5 7 1 4 2 8 6 3

5 7 2 4 8 1 3 6

5 7 2 6 3 1 4 8

5 7 2 6 3 1 8 4

5 7 4 1 3 8 6 2

5 8 4 1 3 6 2 7

5 8 4 1 7 2 6 3

6 1 5 2 8 3 7 4

6 2 7 1 3 5 8 4

6 2 7 1 4 8 5 3

6 3 1 7 5 8 2 4

6 3 1 8 4 2 7 5

6 3 1 8 5 2 4 7

6 3 5 7 1 4 2 8

6 3 5 8 1 4 2 7

6 3 7 2 4 8 1 5

6 3 7 2 8 5 1 4

6 3 7 4 1 8 2 5

6 4 1 5 8 2 7 3

6 4 2 8 5 7 1 3

6 4 7 1 3 5 2 8

6 4 7 1 8 2 5 3

6 8 2 4 1 7 5 3

7 1 3 8 6 4 2 5

7 2 4 1 8 5 3 6

7 2 6 3 1 4 8 5

7 3 1 6 8 5 2 4

7 3 8 2 5 1 6 4

7 4 2 5 8 1 3 6

7 4 2 8 6 1 3 5

7 5 3 1 6 8 2 4

8 2 4 1 7 5 3 6

8 2 5 3 1 7 4 6

8 3 1 6 2 5 7 4

8 4 1 3 6 2 7 5

92

源代码:

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

// 自定义 pair 结构体
template <typename T1, typename T2>
struct MyPair {
    T1 first;
    T2 second;

    MyPair() : first(), second() {}
    MyPair(T1 a, T2 b) : first(a), second(b) {}
};

// 自定义栈结构体
template <typename T>
struct MyStack {
    struct Node {
        T data;
        Node* next;
        Node(T val) : data(val), next(nullptr) {}
    };

    Node* topNode;  // 栈顶元素

    MyStack() : topNode(nullptr) {}

    // 判断栈是否为空
    bool empty() const {
        return topNode == nullptr;
    }

    // 入栈
    void push(const T& value) {
        Node* newNode = new Node(value);
        newNode->next = topNode;
        topNode = newNode;
    }

    // 出栈
    void pop() {
        if (!empty()) {
```

```cpp
            Node* temp = topNode;
            topNode = topNode->next;
           // delete temp;
        }
    }

    // 获取栈顶元素
    T top() const {
        if (!empty()) {
            return topNode->data;
        }
        throw runtime_error("Stack is empty");
    }

    // 析构栈，释放内存
    ~MyStack() {
        while (!empty()) {
            pop();
        }
    }
};

// 定义常量和数组
int n = 8, ans = 0;
int d[50] = {0}, ud[50] = {0}, b[50] = {0};
MyStack<MyPair<int, int>> st;  // 使用自定义栈类型

void f() {
    int step = 1;
    int col = 1;
    while (true) {
        bool found = false;

        for (; col <= n; col++) {
            if (d[col] == 0 && ud[col + step - 1] == 0 && b[col - step + n - 1] == 0) {
                d[col] = 1;
                ud[col + step - 1] = 1;
                b[col - step + n - 1] = 1;
                st.push(MyPair<int, int>(step, col));
                step++;
                col = 1;
                found = true;
                break;
```

```
            }
        }

        if (!found) {
            if (st.empty()) break;
            MyPair<int, int> last = st.top();
            st.pop();
            step = last.first;
            col = last.second;
            d[col] = 0;
            ud[col + step - 1] = 0;
            b[col - step + n - 1] = 0;
            col++;
        } else if (step == n + 1) {
            if (ans < 100) {
                MyStack<MyPair<int, int>> temp = st;
                int s[20] = {0};
                while (!temp.empty()) {
                    s[temp.top().first] = temp.top().second;
                    temp.pop();
                }
                for (int i = 1; i <= n; i++) {
                    cout << s[i] << " ";
                }
                cout << endl;
            }
            ans++;
            MyPair<int, int> last = st.top();
            st.pop();
            step = last.first;
            col = last.second;
            d[col] = 0;
            ud[col + step - 1] = 0;
            b[col - step + n - 1] = 0;
            col++;
        }
    }
    cout << ans;
}

int main() {
    f();
    return 0;
```

```
}
```

3.家谱管理系统

题目:

建立一个家谱管理系统,可以添加,删除,修改,查询,显示家谱.

数据结构:

为每一个成员设立**id**指针,建立索引实现快速快速查找,再建立**vector**来存储孩子节点.实现类似**链表**的数据结构存储.

```
class Member {
public:
    int id;                     // 标识符
    string name;                // 姓名
    string birth_date;          // 出生日期
    string marital_status;      // 婚否
    string address;             // 地址
    bool is_alive;              // 是否健在
    string death_date;          // 死亡日期
    int parent_id;              // 父亲的ID (0表示无)
    vector<int> children_ids;   // 孩子的ID列表
};
```

算法思想:

使用C++的**fstream**库来读取和写入文件,利用id来唯一标识家庭成员,建立索引,实现伪链表的数据结构.
利用**bfs**判断第n代,进行缩进输出.

测试结果:

```
=== 家谱管理系统 ===
1. 添加成员
2. 删除成员
3. 修改成员信息
4. 按姓名查询成员信息
5. 按出生日期查询成员名单
6. 确定两人关系
7. 添加孩子
8. 显示第n代所有人
9. 生成示例数据
a. 打印家族树
0. 退出
请选择功能 (0-10): 8

--- 显示第n代所有人 ---
请输入要显示的代数 n: 2
第 2 代成员:
ID: 4, 姓名: 父亲2, 出生日期: 1967-04-04, 地址: 地址B
ID: 3, 姓名: 母亲222, 出生日期: 1965-03-03, 地址: 地址B

=== 家谱管理系统 ===
1. 添加成员
2. 删除成员
3. 修改成员信息
4. 按姓名查询成员信息
5. 按出生日期查询成员名单
6. 确定两人关系
7. 添加孩子
8. 显示第n代所有人
9. 生成示例数据
a. 打印家族树
0. 退出
请选择功能 (0-10): a

--- 打印家族树 ---
家族成员树形图:
├─ 祖父2 (ID:2, 生日:1942-02-02, 已故)
    ├─ 父亲2 (ID:4, 生日:1967-04-04, 在世)
        ├─ 女儿1 (ID:15, 生日:1992-06-28, 在世)
            ├─ 孙子111 (ID:45, 生日:2015-07-28, 在世)
            ├─ 孙子112 (ID:46, 生日:2015-07-28, 在世)
```

```
        ├ 女儿2 (ID:16, 生日:1992-06-28, 在世)
            ├ 孙子121 (ID:47, 生日:2015-07-28, 在世)
            ├ 孙子122 (ID:48, 生日:2015-07-28, 在世)
        ├ 女儿3 (ID:17, 生日:1992-06-28, 在世)
            ├ 孙子131 (ID:49, 生日:2015-07-28, 在世)
            ├ 孙子132 (ID:50, 生日:2015-07-28, 在世)
        ├ 女儿4 (ID:18, 生日:1992-06-28, 在世)
            ├ 孙子141 (ID:51, 生日:2015-07-28, 在世)
            ├ 孙子142 (ID:52, 生日:2015-07-28, 在世)
        ├ 女儿5 (ID:19, 生日:1992-06-28, 在世)
            ├ 孙子151 (ID:53, 生日:2015-07-28, 在世)
            ├ 孙子152 (ID:54, 生日:2015-07-28, 在世)
        ├ 女儿6 (ID:20, 生日:1992-06-28, 在世)
            ├ 孙子161 (ID:55, 生日:2015-07-28, 在世)
            ├ 孙子162 (ID:56, 生日:2015-07-28, 在世)
        ├ 女儿7 (ID:21, 生日:1992-06-28, 在世)
            ├ 孙子171 (ID:57, 生日:2015-07-28, 在世)
            ├ 孙子172 (ID:58, 生日:2015-07-28, 在世)
        ├ 女儿8 (ID:22, 生日:1992-06-28, 在世)
            ├ 孙子181 (ID:59, 生日:2015-07-28, 在世)
            ├ 孙子182 (ID:60, 生日:2015-07-28, 在世)
        ├ 女儿9 (ID:23, 生日:1992-06-28, 在世)
            ├ 孙子191 (ID:61, 生日:2015-07-28, 在世)
            ├ 孙子192 (ID:62, 生日:2015-07-28, 在世)
        ├ 女儿10 (ID:24, 生日:1992-06-28, 在世)
            ├ 孙子201 (ID:63, 生日:2015-07-28, 在世)
            ├ 孙子202 (ID:64, 生日:2015-07-28, 在世)
├ 祖父1 (ID:1, 生日:1940-01-01, 已故)
    ├ 母亲222 (ID:3, 生日:1965-03-03, 在世)
        ├ 儿子1 (ID:5, 生日:1990-05-27, 在世)
            ├ 孙子11 (ID:25, 生日:2015-07-28, 在世)
            ├ 孙子12 (ID:26, 生日:2015-07-28, 在世)
        ├ 儿子2 (ID:6, 生日:1990-05-27, 在世)
            ├ 孙子21 (ID:27, 生日:2015-07-28, 在世)
            ├ 孙子22 (ID:28, 生日:2015-07-28, 在世)
        ├ 儿子3 (ID:7, 生日:1990-05-27, 在世)
            ├ 孙子31 (ID:29, 生日:2015-07-28, 在世)
            ├ 孙子32 (ID:30, 生日:2015-07-28, 在世)
        ├ 儿子4 (ID:8, 生日:1990-05-27, 在世)
            ├ 孙子41 (ID:31, 生日:2015-07-28, 在世)
            ├ 孙子42 (ID:32, 生日:2015-07-28, 在世)
        ├ 儿子5 (ID:9, 生日:1990-05-27, 在世)
            ├ 孙子51 (ID:33, 生日:2015-07-28, 在世)
```

```
            ├─ 孙子52 (ID:34, 生日:2015-07-28, 在世)
        ├─ 儿子6 (ID:10, 生日:1990-05-27, 在世)
            ├─ 孙子61 (ID:35, 生日:2015-07-28, 在世)
            ├─ 孙子62 (ID:36, 生日:2015-07-28, 在世)
        ├─ 儿子7 (ID:11, 生日:1990-05-27, 在世)
            ├─ 孙子71 (ID:37, 生日:2015-07-28, 在世)
            ├─ 孙子72 (ID:38, 生日:2015-07-28, 在世)
        ├─ 儿子8 (ID:12, 生日:1990-05-27, 在世)
            ├─ 孙子81 (ID:39, 生日:2015-07-28, 在世)
            ├─ 孙子82 (ID:40, 生日:2015-07-28, 在世)
        ├─ 儿子9 (ID:13, 生日:1990-05-27, 在世)
            ├─ 孙子91 (ID:41, 生日:2015-07-28, 在世)
            ├─ 孙子92 (ID:42, 生日:2015-07-28, 在世)
        ├─ 儿子10 (ID:14, 生日:1990-05-27, 在世)
            ├─ 孙子101 (ID:43, 生日:2015-07-28, 在世)
            ├─ 孙子102 (ID:44, 生日:2015-07-28, 在世)
├─ s (ID:65, 生日:s, 已故)
├─ a (ID:66, 生日:a, 已故)
```

源代码:

```cpp
#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>
#include <fstream>
#include <sstream>
#include <algorithm>
#include <queue>

using namespace std;

class Member {
public:
    int id;                     // 标识符
    string name;                // 姓名
    string birth_date;          // 出生日期
    string marital_status;      // 婚否
    string address;             // 地址
    bool is_alive;              // 是否健在
    string death_date;          // 死亡日期
    int parent_id;              // 父亲的ID (0表示无)
    vector<int> children_ids;   // 孩子的ID列表

    Member() : id(0), parent_id(0), is_alive(true) {}

    // 将Member对象转换为文件
    string to_string_line() const {
        string line = to_string(id) + "|" + name + "|" + birth_date + "|" + marital_status + "|'
                      (is_alive ? "1" : "0") + "|" + death_date + "|" + to_string(parent_id) + '
        for (auto i = 0; i < children_ids.size(); ++i) {
            line += to_string(children_ids[i]);
            if (i != children_ids.size() - 1)
                line += ";";
        }
        return line;
    }

    // 从文件创建Member对象
    static Member from_string_line(const string& line) {
        Member member;
        stringstream ss(line);
```

```cpp
    string item;
    vector<string> tokens;

    while (getline(ss, item, '|')) {
        tokens.push_back(item);
    }

    if (tokens.size() < 8) {
        cerr << "数据格式错误: " << line << endl;
        return member;
    }

    member.id = stoi(tokens[0]);
    member.name = tokens[1];
    member.birth_date = tokens[2];
    member.marital_status = tokens[3];
    member.address = tokens[4];
    member.is_alive = (tokens[5] == "1") ? true : false;
    member.death_date = tokens[6];
    member.parent_id = stoi(tokens[7]);

    if (tokens.size() >= 9 && !tokens[8].empty()) {
        string children_str = tokens[8];
        stringstream cs(children_str);
        string child_id_str;
        while (getline(cs, child_id_str, ';')) {
            member.children_ids.push_back(stoi(child_id_str));
        }
    }

    return member;
}

// 打印成员信息
void print_info(const unordered_map<int, Member>& members_map) const {
    cout << "ID: " << id << endl;
    cout << "姓名: " << name << endl;
    cout << "出生日期: " << birth_date << endl;
    cout << "婚否: " << marital_status << endl;
    cout << "地址: " << address << endl;
    cout << "健在否: " << (is_alive ? "是" : "否") << endl;
    if (!is_alive) {
        cout << "死亡日期: " << death_date << endl;
```

```
        }
        // 父亲信息
        if (parent_id != 0) {
            auto it = members_map.find(parent_id);
            if (it != members_map.end()) {
                cout << "父亲信息：" << endl;
                cout << "  姓名：" << it->second.name << endl;
                cout << "  出生日期：" << it->second.birth_date << endl;
            }
        }
        // 孩子信息
        if (!children_ids.empty()) {
            cout << "孩子信息：" << endl;
            for (int child_id : children_ids) {
                auto it = members_map.find(child_id);
                if (it != members_map.end()) {
                    cout << "  姓名：" << it->second.name << ", 出生日期：" << it->second.birth_
                }
            }
        }
    }
};


class GenealogyManager {
private:
    unordered_map<int, Member> members_map; // 哈希:key: id, value: Member
    string file_path;
    int next_id;

public:
    GenealogyManager(const string& path = "genealogy.txt") : file_path(path), next_id(1) {
        load_data();
    }

    // 加载数据从文件
    void load_data() {
        ifstream infile(file_path);
        if (!infile.is_open()) {
            // 文件不存在，初始化为空
            cout << "数据文件不存在。" << endl;
            return;
        }
```

```
    string line;
    while (getline(infile, line)) {
        if (line.empty()) continue;
        Member member = Member::from_string_line(line);
        members_map[member.id] = member;
        if (member.id >= next_id) {
            next_id = member.id + 1;
        }
    }

    infile.close();
}

// 保存数据到文件
void save_data() const {
    ofstream outfile(file_path, ios::trunc);
    if (!outfile.is_open()) {
        cerr << "无法打开文件进行写入: " << file_path << endl;
        return;
    }

    for (const auto& pair : members_map) {
        outfile << pair.second.to_string_line() << "\n";
    }

    outfile.close();
}

// 添加成员
void add_member(Member& member) {
    member.id = next_id++;
    members_map[member.id] = member;

    // 更新父亲的孩子列表
    if (member.parent_id != 0) {
        auto it = members_map.find(member.parent_id);
        if (it != members_map.end()) {
            it->second.children_ids.push_back(member.id);
        } else {
            cout << "警告: 父亲ID " << member.parent_id << " 未找到。" << endl;
        }
    }
```

```
    save_data();
    cout << "成员 '" << member.name << "' 已添加，ID为 " << member.id << "。\n";
}

// 删除成员及其后代
void delete_member(int member_id) {
    auto it = members_map.find(member_id);
    if (it == members_map.end()) {
        cout << "成员ID " << member_id << " 未找到。\n";
        return;
    }

    // 递归删除所有后代
    for (int child_id : it->second.children_ids) {
        delete_member(child_id);
    }

    // 从父亲的孩子列表中移除
    if (it->second.parent_id != 0) {
        auto parent_it = members_map.find(it->second.parent_id);
        if (parent_it != members_map.end()) {
            parent_it->second.children_ids.erase(
                remove(parent_it->second.children_ids.begin(), parent_it->second.children_i
                parent_it->second.children_ids.end()
            );
        }
    }

    // 删除成员
    members_map.erase(it);
    cout << "成员ID " << member_id << " 及其后代已删除。\n";

    save_data();
}

// 修改成员信息
void modify_member(int member_id, const Member& updated_member) {
    auto it = members_map.find(member_id);
    if (it == members_map.end()) {
        cout << "成员ID " << member_id << " 未找到。\n";
        return;
    }
```

```
        // 如果父亲ID发生变化，需要更新原父亲和新父亲的孩子列表
        if (updated_member.parent_id != it->second.parent_id) {
            // 从原父亲的孩子列表中移除
            if (it->second.parent_id != 0) {
                auto original_parent_it = members_map.find(it->second.parent_id);
                if (original_parent_it != members_map.end()) {
                    original_parent_it->second.children_ids.erase(
                        remove(original_parent_it->second.children_ids.begin(), original_parent_
                        original_parent_it->second.children_ids.end()
                    );
                }
            }

            // 添加到新父亲的孩子列表
            if (updated_member.parent_id != 0) {
                auto new_parent_it = members_map.find(updated_member.parent_id);
                if (new_parent_it != members_map.end()) {
                    new_parent_it->second.children_ids.push_back(member_id);
                } else {
                    cout << "警告: 新父亲ID " << updated_member.parent_id << " 未找到。\n";
                }
            }
        }

        // 更新成员信息
        it->second.name = updated_member.name;
        it->second.birth_date = updated_member.birth_date;
        it->second.marital_status = updated_member.marital_status;
        it->second.address = updated_member.address;
        it->second.is_alive = updated_member.is_alive;
        it->second.death_date = updated_member.death_date;
        it->second.parent_id = updated_member.parent_id;
        // children_ids 不在此处更新

        save_data();
        cout << "成员ID " << member_id << " 信息已更新。\n";
    }

    // 查询成员通过姓名（可能有多个同名成员）
    vector<Member> get_members_by_name(const string& name) const {
        vector<Member> result;
        for (const auto& pair : members_map) {
```

```
        if (pair.second.name == name) {
            result.push_back(pair.second);
        }
    }
    return result;
}

// 查询成员通过出生日期
vector<Member> get_members_by_birth_date(const string& birth_date) const {
    vector<Member> result;
    for (const auto& pair : members_map) {
        if (pair.second.birth_date == birth_date) {
            result.push_back(pair.second);
        }
    }
    return result;
}

// 获取第n代成员
vector<Member>  get_nth_generation(int n) const {
    vector<Member> generation;
    if (n < 1) return generation;

    // 获取根代（无父亲）
    vector<Member> current_gen;
    for (const auto& pair : members_map) {
        if (pair.second.parent_id == 0) {
            current_gen.push_back(pair.second);
        }
    }

    int current_level = 1;
    while (current_level < n && !current_gen.empty()) {
        vector<Member> next_gen;
        for (const Member& member : current_gen) {
            for (int child_id : member.children_ids) {
                auto it = members_map.find(child_id);
                if (it != members_map.end()) {
                    next_gen.push_back(it->second);
                }
            }
        }
        current_gen = next_gen;
```

```
        current_level++;
    }

    if (current_level == n) {
        generation = current_gen;
    }

    return generation;
}

// 获取成员的所有祖先
vector<Member> get_ancestors(int member_id) const {
    vector<Member> ancestors;
    auto it = members_map.find(member_id);
    while (it != members_map.end() && it->second.parent_id != 0) {
        auto parent_it = members_map.find(it->second.parent_id);
        if (parent_it != members_map.end()) {
            ancestors.push_back(parent_it->second);
            it = parent_it;
        } else {
            break;
        }
    }
    return ancestors;
}

// 确定两人关系
string determine_relationship(int id1, int id2) const {
    if (id1 == id2) return "同一个人。";

    // 获取所有祖先
    vector<Member> ancestors1 = get_ancestors(id1);
    vector<Member> ancestors2 = get_ancestors(id2);

    // 寻找共同祖先
    int common_ancestor_id = 0;
    for (const Member& anc1 : ancestors1) {
        for (const Member& anc2 : ancestors2) {
            if (anc1.id == anc2.id) {
                common_ancestor_id = anc1.id;
                break;
            }
        }
```

```
        if (common_ancestor_id != 0) break;
    }

    if (common_ancestor_id == 0) {
        return "无共同祖先，关系不明确。";
    } else {
        // 简单描述
        auto it = members_map.find(common_ancestor_id);
        if (it != members_map.end()) {
            return "两人有共同的祖先：" + it->second.name + "。具体关系需要进一步分析。";
        } else {
            return "两人有共同的祖先。具体关系需要进一步分析。";
        }
    }
}

// 添加孩子
void add_child(int parent_id, const Member& child) {
    auto it = members_map.find(parent_id);
    if (it == members_map.end()) {
        cout << "父亲ID " << parent_id << " 未找到。\n";
        return;
    }

    Member new_child = child;
    new_child.id = next_id++;
    new_child.parent_id = parent_id;
    members_map[new_child.id] = new_child;
    it->second.children_ids.push_back(new_child.id);

    save_data();
    cout << "孩子 '" << new_child.name << "' 已添加，ID为 " << new_child.id << "。\n";
}

// 打印成员信息
void print_member_info(int member_id) const {
    auto it = members_map.find(member_id);
    if (it == members_map.end()) {
        cout << "成员ID " << member_id << " 未找到。\n";
        return;
    }
    it->second.print_info(members_map);
}
```

```cpp
// 生成示例数据
void generate_sample_data() {
    cout << "正在生成示例数据...\n";

    // 创建祖先
    Member grandfather;
    grandfather.name = "祖父1";
    grandfather.birth_date = "1940-01-01";
    grandfather.marital_status = "已婚";
    grandfather.address = "地址A";
    grandfather.is_alive = false;
    grandfather.death_date = "2000-05-05";
    grandfather.parent_id = 0;
    add_member(grandfather); // ID 1

    Member grandmother;
    grandmother.name = "祖父2";
    grandmother.birth_date = "1942-02-02";
    grandmother.marital_status = "已婚";
    grandmother.address = "地址A";
    grandmother.is_alive = false;
    grandmother.death_date = "2005-06-06";
    grandmother.parent_id = 0;
    add_member(grandmother); // ID 2

    // 创建父母
    Member father;
    father.name = "父亲1";
    father.birth_date = "1965-03-03";
    father.marital_status = "已婚";
    father.address = "地址B";
    father.is_alive = true;
    father.parent_id = grandfather.id;
    add_member(father); // ID 3

    Member mother;
    mother.name = "父亲2";
    mother.birth_date = "1967-04-04";
    mother.marital_status = "已婚";
    mother.address = "地址B";
    mother.is_alive = true;
    mother.parent_id = grandmother.id;
```

```
    add_member(mother); // ID 4

    // 创建子女
    for (int i = 1; i <= 10; ++i) {
        Member son;
        son.name = "儿子" + to_string(i);
        son.birth_date = "1990-05-27";
        son.marital_status = "未婚";
        son.address = "地址C";
        son.is_alive = true;
        son.parent_id = father.id;
        add_member(son); // IDs 5-14
    }

    for (int i = 1; i <= 10; ++i) {
        Member daughter;
        daughter.name = "女儿" + to_string(i);
        daughter.birth_date = "1992-06-28" ;
        daughter.marital_status = "未婚";
        daughter.address = "地址C";
        daughter.is_alive = true;
        daughter.parent_id = mother.id;
        add_member(daughter); // IDs 15-24
    }

    // 创建孙辈
    for (int i = 1; i <= 20; ++i) {
        // 每个儿子有2个孩子
        for (int j = 1; j <= 2; ++j) {
            Member grandchild;
            grandchild.name = "孙子" + to_string(i) + to_string(j);
            grandchild.birth_date = "2015-07-28";
            grandchild.marital_status = "未婚";
            grandchild.address = "地址D";
            grandchild.is_alive = true;
            // 假设儿子i的ID是4 + i (根据上面的添加顺序)
            grandchild.parent_id = 4 + i;
            add_member(grandchild); // IDs 25-44
        }
    }

    cout << "示例数据已生成，共 " << members_map.size() << " 个成员。\n";
}
```

```cpp
    // 获取下一个唯一ID（用于确保唯一性）
    int get_next_id() const {
        return next_id;
    }

    void print_family_tree(int root_id = 0, int depth = 0) const {
        // 如果是打印整个家谱树（root_id = 0），则找出所有根节点（没有父亲的成员）
        if (root_id == 0) {
            cout << "家族成员树形图：\n";
            for (const auto& pair : members_map) {
                if (pair.second.parent_id == 0) {
                    print_family_tree(pair.first, 0);
                }
            }
            return;
        }

        // 打印当前成员
        auto it = members_map.find(root_id);
        if (it == members_map.end()) return;

        // 打印缩进
        for (int i = 0; i < depth; i++) {
            cout << "    ";
        }

        // 打印成员信息
        cout << "├ " << it->second.name
             << " (ID:" << it->second.id
             << ", 生日:" << it->second.birth_date
             << ", " << (it->second.is_alive ? "在世" : "已故")
             << ")\n";

        // 递归打印所有子女
        for (int child_id : it->second.children_ids) {
            print_family_tree(child_id, depth + 1);
        }
    }
};

// 读取整数输入，带错误检查
int read_int(const string& prompt) {
```

```cpp
    int value;
    while (true) {
        cout << prompt;
        string input;
        getline(cin, input);
        try {
            value = stoi(input);
            break;
        } catch (...) {
            cout << "无效输入，请输入一个整数。\n";
        }
    }
    return value;
}

// 选择成员（处理同名情况）
int select_member(const vector<Member>& members) {
    if (members.empty()) {
        return -1;
    } else if (members.size() == 1) {
        return members[0].id;
    } else {
        cout << "找到多个同名成员，请选择：\n";
        for (size_t i = 0; i < members.size(); ++i) {
            cout << i + 1 << ". ID: " << members[i].id << ", 出生日期: " << members[i].birth_dat
        }
        int choice = 0;
        while (true) {
            choice = read_int("选择编号: ");
            if (choice >= 1 && choice <= members.size()) {
                return members[choice - 1].id;
            } else {
                cout << "无效选择，请重新输入。\n";
            }
        }
    }
}


// 添加成员
void add_member_ui(GenealogyManager& manager) {
    cout << "\n--- 添加成员 ---\n";
    Member member;
```

```
    cout << "姓名: ";
    cin >> member.name;
    cout << "出生日期 (YYYY-MM-DD): ";
    cin >> member.birth_date;
    cout << "婚否: ";
    cin >> member.marital_status;
    cout << "地址: ";
    cin >> member.address;
    cout << "是否健在 (1是/0否): ";
    string alive_input;
    cin >> alive_input;
    member.is_alive = (alive_input == "1") ? true : false;
    if (!member.is_alive) {
        cout << "死亡日期 (YYYY-MM-DD): ";
        cin >> member.death_date;
    } else {
        member.death_date = "";
    }
    cout << "父亲ID (0表示无): ";
    string parent_input;
    cin >> parent_input;
    try {
        member.parent_id = stoi(parent_input);
    } catch (...) {
        member.parent_id = 0;
    }

    manager.add_member(member);
}

// 删除成员
void delete_member_ui(GenealogyManager& manager) {
    cout << "\n--- 删除成员 ---\n";
    cout << "请输入要删除的成员姓名: ";
    string name;
    cin >> name;
    vector<Member> members = manager.get_members_by_name(name);
    if (members.empty()) {
        cout << "未找到该成员。\n";
        return;
    }
    int member_id = select_member(members);
    if (member_id != -1) {
```

```cpp
        cout << "确定要删除成员ID " << member_id << " 及其后代吗？ (y/n): ";
        string confirm;
        cin >> confirm;
        if (confirm == "y" || confirm == "Y") {
            manager.delete_member(member_id);
        } else {
            cout << "取消删除。\n";
        }
    }
}

// 修改成员信息
void modify_member_ui(GenealogyManager& manager) {
    cout << "\n--- 修改成员信息 ---\n";
    cout << "请输入要修改的成员姓名: ";
    string name;
    cin >> name;
    vector<Member> members = manager.get_members_by_name(name);
    if (members.empty()) {
        cout << "未找到该成员。\n";
        return;
    }
    int member_id = select_member(members);
    if (member_id == -1) return;

    // 获取当前成员信息
    Member current_member;
    auto it = manager.get_members_by_name(name).begin();
    for (const Member& m : members) {
        if (m.id == member_id) {
            current_member = m;
            break;
        }
    }

    // 输入新的信息，按回车跳过不修改
    cout << "按回车跳过不修改。\n";
    cout << "姓名 [" << current_member.name << "]: ";
    string input;
    cin >> input;
    if (!input.empty()) current_member.name = input;

    cout << "出生日期 [" << current_member.birth_date << "]: ";
```

```
    cin >> input;
    if (!input.empty()) current_member.birth_date = input;

    cout << "婚否 [" << current_member.marital_status << "]: ";
    cin >> input;
    if (!input.empty()) current_member.marital_status = input;

    cout << "地址 [" << current_member.address << "]: ";
    cin >> input;
    if (!input.empty()) current_member.address = input;

    cout << "是否健在 (1是/0否) [" << (current_member.is_alive ? "1" : "0") << "]: ";
    cin >> input;
    if (!input.empty()) {
        current_member.is_alive = (input == "1") ? true : false;
    }

    if (!current_member.is_alive) {
        cout << "死亡日期 [" << (current_member.death_date.empty() ? "无" : current_member.death
        cin >> input;
        if (!input.empty()) current_member.death_date = input;
    } else {
        current_member.death_date = "";
    }

    cout << "父亲ID [" << current_member.parent_id << "]: ";
    cin >> input;
    if (!input.empty()) {
        try {
            current_member.parent_id = stoi(input);
        } catch (...) {
            current_member.parent_id = 0;
        }
    }

    manager.modify_member(member_id, current_member);
}

// 按姓名查询成员信息
void query_by_name_ui(const GenealogyManager& manager) {
    cout << "\n--- 按姓名查询成员信息 ---\n";
    cout << "请输入要查询的成员姓名: ";
    string name;
```

```
    cin >> name;
    vector<Member> members = manager.get_members_by_name(name);
    if (members.empty()) {
        cout << "未找到该成员。\n";
        return;
    }
    for (const Member& member : members) {
        cout << "--------------------------\n";
        const_cast<GenealogyManager&>(manager).print_member_info(member.id);
    }
    cout << "--------------------------\n";
}

// 按出生日期查询成员名单
void query_by_birth_date_ui(const GenealogyManager& manager) {
    cout << "\n--- 按出生日期查询成员名单 ---\n";
    cout << "请输入出生日期 (YYYY-MM-DD): ";
    string birth_date;
    cin >> birth_date;
    vector<Member> members = manager.get_members_by_birth_date(birth_date);
    if (members.empty()) {
        cout << "未找到符合出生日期的成员。\n";
        return;
    }
    cout << "找到以下成员:\n";
    for (const Member& member : members) {
        cout << "ID: " << member.id << ", 姓名: " << member.name << ", 地址: " << member.address
    }
}

// 确定两人关系
void determine_relationship_ui(const GenealogyManager& manager) {
    cout << "\n--- 确定两人关系 ---\n";
    cout << "请输入第一个成员姓名: ";
    string name1;
    cin >> name1;
    cout << "请输入第二个成员姓名: ";
    string name2;
    cin >> name2;

    vector<Member> members1 = manager.get_members_by_name(name1);
    vector<Member> members2 = manager.get_members_by_name(name2);
```

```cpp
    if (members1.empty() || members2.empty()) {
        cout << "其中一位成员未找到。\n";
        return;
    }

    int id1 = select_member(members1);
    int id2 = select_member(members2);

    if (id1 == -1 || id2 == -1) return;

    string relation = manager.determine_relationship(id1, id2);
    cout << "关系: " << relation << "\n";
}

// 添加孩子
void add_child_ui(GenealogyManager& manager) {
    cout << "\n--- 添加孩子 ---\n";
    cout << "请输入父亲姓名: ";
    string parent_name;
    cin >> parent_name;
    vector<Member> parents = manager.get_members_by_name(parent_name);
    if (parents.empty()) {
        cout << "未找到该父亲。\n";
        return;
    }
    int parent_id = select_member(parents);
    if (parent_id == -1) return;

    // 输入孩子信息
    Member child;
    cout << "孩子姓名: ";
    cin >> child.name;
    cout << "孩子出生日期 (YYYY-MM-DD): ";
    cin >> child.birth_date;
    cout << "孩子婚否: ";
    cin >> child.marital_status;
    cout << "孩子地址: ";
    cin >> child.address;
    cout << "孩子是否健在 (1是/0否): ";
    string alive_input;
    cin >> alive_input;
    child.is_alive = (alive_input == "1") ? true : false;
    if (!child.is_alive) {
```

```
        cout << "孩子死亡日期 (YYYY-MM-DD): ";
        cin >> child.death_date;
    } else {
        child.death_date = "";
    }

    manager.add_child(parent_id, child);
}

// 显示第n代所有人
void display_nth_generation_ui(const GenealogyManager& manager) {
    cout << "\n--- 显示第n代所有人 ---\n";
    cout<<("请输入要显示的代数 n: ");
    int n;
    cin >> n;
    if (n < 1) {
        cout << "代数必须大于等于1。\n";
        return;
    }
    vector<Member> generation = manager.get_nth_generation(n);
    if (generation.empty()) {
        cout << "第 " << n << " 代无成员。\n";
    } else {
        cout << "第 " << n << " 代成员:\n";
        for (const Member& member : generation) {
            cout << "ID: " << member.id << ", 姓名: " << member.name << ", 出生日期: " << member
        }
    }
}

// 生成示例数据
void generate_sample_data_ui(GenealogyManager& manager) {
    cout << "\n--- 生成示例数据 ---\n";
    manager.generate_sample_data();
}

// 打印家族树
void print_family_tree_ui(const GenealogyManager& manager) {
    cout << "\n--- 打印家族树 ---\n";
    manager.print_family_tree();
}

// 主菜单
```

```cpp
void display_menu() {
    cout << "\n=== 家谱管理系统 ===\n";
    cout << "1. 添加成员\n";
    cout << "2. 删除成员\n";
    cout << "3. 修改成员信息\n";
    cout << "4. 按姓名查询成员信息\n";
    cout << "5. 按出生日期查询成员名单\n";
    cout << "6. 确定两人关系\n";
    cout << "7. 添加孩子\n";
    cout << "8. 显示第n代所有人\n";
    cout << "9. 生成示例数据\n";
    cout << "a. 打印家族树\n";
    cout << "0. 退出\n";
    cout << "请选择功能 (0-10): ";
}

// --- Main Function ---
int main() {
    GenealogyManager manager;
    while (true) {
        display_menu();
        string choice;
        cin >> choice;
        if (choice.empty()) continue;
        switch (choice[0]) {
            case '1':
                add_member_ui(manager);
                break;
            case '2':
                delete_member_ui(manager);
                break;
            case '3':
                modify_member_ui(manager);
                break;
            case '4':
                query_by_name_ui(manager);
                break;
            case '5':
                query_by_birth_date_ui(manager);
                break;
            case '6':
                determine_relationship_ui(manager);
                break;
```

```
            case '7':
                add_child_ui(manager);
                break;
            case '8':
                display_nth_generation_ui(manager);
                break;
            case '9':
                generate_sample_data_ui(manager);
                break;
            case 'a':
                print_family_tree_ui(manager);
                break;
            case '0':
                cout << "退出系统。\n";
                return 0;
            default:
                cout << "无效选择，请重新输入。\n";
        }
    }
}
```

4.平衡二叉树

题目:

对于1-10000的质数序列<2，3，5，7，…，9973>，建立平衡二叉排序树。

数据结构:

建立结构体**Node**来存储节点信息,再建立**vector**来存储各个节点.\

```
struct Node {
    int data;
    int height;  // 节点的高度
    Node* left;
    Node* right;
    Node(int val) : data(val), height(0), left(nullptr), right(nullptr) {}
};
```

算法思想:

旋转即为将根节点的子节点作为根节点,再由子节点的子节点(假设为孙节点)作为根节点的子节点.
再根据根节点及其子节点的平衡因子判断如何旋转.删除和插入类似.

测试结果:

```
211 yes
223 yes
227 yes
229 yes
233 yes
239 yes
241 yes
251 yes
257 yes
263 yes
269 yes
271 yes
277 yes
281 yes
283 yes
293 yes


601 no
607 no
613 no
617 no
619 no
631 no
641 no
643 no
647 no
653 no
659 no
661 no
673 no
677 no
683 no
691 no
```

100 yes

102 yes

104 yes

106 yes

108 yes

110 yes

112 yes

114 yes

116 yes

118 yes

120 yes

122 yes

124 yes

126 yes

128 yes

130 yes

132 yes

134 yes

136 yes

138 yes

140 yes

142 yes

144 yes

146 yes

148 yes

150 yes

152 yes

154 yes

156 yes

158 yes

160 yes

162 yes

164 yes

166 yes

168 yes

170 yes

172 yes

174 yes

176 yes

178 yes

180 yes

182 yes

184 yes

186 yes

188 yes

190 yes

192 yes

194 yes

196 yes

198 yes

200 yes

源代码:

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;

// 定义AVL树的节点结构
struct Node {
    int data;
    int height;  // 节点的高度
    Node* left;
    Node* right;
    Node(int val) : data(val), height(0), left(nullptr), right(nullptr) {}
};

// 获取节点高度
int height(Node* node) {
    return node == nullptr ? -1 : node->height;
}

// 更新节点的高度
void updateHeight(Node* &node) {
    if (node == nullptr) return;
    int leftHeight = (node->left != nullptr) ? node->left->height : -1;
    int rightHeight = (node->right != nullptr) ? node->right->height : -1;
    node->height = max(leftHeight, rightHeight) + 1;
}

// 获取平衡因子
int balanceFactor(Node* node) {
    return height(node->left) - height(node->right);
}

// 左旋操作
void leftRotate(Node*& root) {
    Node* child = root->right;
    if (child == nullptr) return;
    Node* grandchild = child->left;
    child->left = root;
    root->right = grandchild;
    updateHeight(root);
```

```
    updateHeight(child);
    root = child;
}

// 右旋操作
void rightRotate(Node*& root) {
    Node* child = root->left;
    Node* grandchild = child->right;
    child->right = root;
    root->left = grandchild;
    updateHeight(root);
    updateHeight(child);
    root = child;
}

// 插入操作
void insert(Node*& root, int value) {
    if (root == nullptr) {
        root = new Node(value);
        return;
    }
    if (value < root->data) {
        insert(root->left, value);
    } else if (value > root->data) {
        insert(root->right, value);
    }

    updateHeight(root);

    int balance = balanceFactor(root);

    // 左左情况
    if (balance > 1 && value < root->left->data) {
        rightRotate(root);
    }
    // 右右情况
    if (balance < -1 && value > root->right->data) {
        leftRotate(root);
    }
    // 左右情况
    if (balance > 1 && value > root->left->data) {
        leftRotate(root->left);
        rightRotate(root);
```

```
    }
    // 右左情况
    if (balance < -1 && value < root->right->data) {
        rightRotate(root->right);
        leftRotate(root);
    }
}

// 删除节点
Node* deleteNode(Node* root, int value) {
    if (root == nullptr) return root;

    if (value < root->data) {
        root->left = deleteNode(root->left, value);
    } else if (value > root->data) {
        root->right = deleteNode(root->right, value);
    } else {
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        } else {
            Node* temp = root->right;
            while (temp && temp->left != nullptr) {
                temp = temp->left;
            }
            root->data = temp->data;
            root->right = deleteNode(root->right, temp->data);
        }
    }

    updateHeight(root);

    int balance = balanceFactor(root);

    // 左左情况
    if (balance > 1 && balanceFactor(root->left) >= 0) {
        rightRotate(root);
    }
```

```
    // 右右情况
    if (balance < -1 && balanceFactor(root->right) <= 0) {
        leftRotate(root);
    }
    // 左右情况
    if (balance > 1 && balanceFactor(root->left) < 0) {
        leftRotate(root->left);
        rightRotate(root);
    }
    // 右左情况
    if (balance < -1 && balanceFactor(root->right) > 0) {
        rightRotate(root->right);
        leftRotate(root);
    }

    return root;
}

// 查询节点
bool search(Node* root, int value) {
    if (root == nullptr) return false;
    if (value == root->data) return true;
    if (value < root->data) return search(root->left, value);
    return search(root->right, value);
}

// 生成质数

// 写入查询结果到文件
vector<int> Eratosthenes(int m) {
    vector<bool> isPrime(m + 1, true);
    isPrime[0] = isPrime[1] = false;
    for (int i = 2; i * i <= m; ++i) {
        if (isPrime[i]) {
            for (int j = i * i; j <= m; j += i) {
                isPrime[j] = false;
            }
        }
    }

    vector<int> primes;
    for (int i = 2; i <= m; ++i) {
        if (isPrime[i]) {
```

```
            primes.push_back(i);
        }
    }

    return primes;
}

// 从 n 到 m 生成质数并返回
vector<int> generatePrimes(int n, int m) {
    vector<int> primes = Eratosthenes(m);
    // 筛选出 [n, m] 范围内的质数
    vector<int> result;
    for (int prime : primes) {
        if (prime >= n) {
            result.push_back(prime);
        }
    }
    return result;
}
void writeQueryResults(const vector<int>& queries, Node* root, const string& filename) {
    ofstream outfile(filename);
    for (int query : queries) {
        outfile << query << " " << (search(root, query) ? "yes" : "no") << endl;
    }
    outfile.close();
}

int main() {
    // 生成质数
    vector<int> primes = generatePrimes(0,10000);

    // 创建平衡二叉排序树并插入1-10000之间的质数
    Node* root = nullptr;
    for (int prime : primes) {
        insert(root, prime);
    }

    // (1) 查询200-300之间的质数
    vector<int> query1 =generatePrimes(200,300);
    writeQueryResults(query1, root, "tree1.txt");

    // (2) 删除500-2000之间的质数并查询600-700之间的质数
    vector<int> deletePrimes;
```

```cpp
    for (int prime : primes) {
        if (prime >= 500 && prime <= 2000) {
            deletePrimes.push_back(prime);
        }
    }
    for (int prime : deletePrimes) {
        root = deleteNode(root, prime);
    }

    vector<int> query2 = generatePrimes(600,700);
    writeQueryResults(query2, root, "tree2.txt");

    // (3) 插入1-1000之间的偶数并查询100-200之间的偶数
    for (int i = 2; i <= 1000; i += 2) {
        insert(root, i);
    }

    vector<int> query3;
    for (int i = 100; i <= 200; i += 2) {
        query3.push_back(i);
    }
    writeQueryResults(query3, root, "tree3.txt");

    return 0;
}
```

5.哈夫曼编码

题目:

对给定的文本文件进行哈夫曼编码，并生成对应的二进制文件。

数据结构:

建立**HuffmanNode**来存储各个节点,再利用优先队列来储存各个字符的编码排序.

```cpp
struct HuffmanNode {
    char ch;                // 存储字符
    int frequency;          // 字符频率
    HuffmanNode* left;      // 左子节点
    HuffmanNode* right;     // 右子节点

    HuffmanNode(char character, int freq) : ch(character), frequency(freq), left(nullptr), righ

    // 用于优先队列的比较函数，按频率从小到大排序
    struct Compare {
        bool operator()(HuffmanNode* left, HuffmanNode* right) {
            return left->frequency > right->frequency;
        }
    };
};
```

算法思想:

利用**map**统计字符频率,然后构建哈夫曼树.深搜获取哈夫曼编码,写入文件.

测试结果:

```
c: 110111
D: 1101101111
H: 1101101110
F: 1101101101
L: 1101101100
0: 1101100111111
7: 1101100111110
X: 1101100111101
3: 1101100111100
E: 110110011101
Z: 110110011100
j: 1101100110
S: 110110010
T: 11011000
h: 11010
p: 110011
.: 1100101
g: 110001
:: 11000011111
8: 1100001111011
9: 1100001111010
6: 1100001111001
R: 1100001111000
A: 1100001110
-: 1100001101
N: 11000011001
V: 11000011000
w: 1100100
(: 0100010010
M: 010000001
r: 0000
I: 010001000
n: 1001
k: 0100001
W: 010000010
;: 01000000001
x: 0100000110
): 0100000111
s: 0101
K: 010000000001
P: 0100010011
```

```
u: 00010
B: 0100000001
d: 00011
e: 001
U: 010000000000
z: 01000101
b: 0100011
l: 01001
 : 111
o: 0110
J: 10100100101
i: 0111
t: 1000
y: 101000
": 11000010
q: 1010010011
1: 101001001000

: 11011010
Y: 101001001001
C: 10100101
,: 1010011
': 101001000
v: 1100000
m: 101010
f: 101011
a: 1011
```

源代码:

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <unordered_map>
#include <queue>
#include <vector>
#include <bitset>
#include <sstream>
using namespace std;

// 定义Huffman树节点
struct HuffmanNode {
    char ch;                // 存储字符
    int frequency;          // 字符频率
    HuffmanNode* left;      // 左子节点
    HuffmanNode* right;     // 右子节点

    HuffmanNode(char character, int freq) : ch(character), frequency(freq), left(nullptr), righ

    // 用于优先队列的比较函数，按频率从小到大排序
    struct Compare {
        bool operator()(HuffmanNode* left, HuffmanNode* right) {
            return left->frequency > right->frequency;
        }
    };
};

// 递归地为每个字符生成Huffman编码
void generate(HuffmanNode* root, const string& code, unordered_map<char, string>& huffmanCodes)
    if (root == nullptr) {
        return;
    }
    if (root->left == nullptr && root->right == nullptr) {
        huffmanCodes[root->ch] = code;
    }
    generate(root->left, code + "0", huffmanCodes);
    generate(root->right, code + "1", huffmanCodes);
}

// 读取文件内容并统计字符频率
unordered_map<char, int> calculate(const string& filename) {
```

```
    ifstream file(filename);
    unordered_map<char, int> freqMap;
    char ch;

    while (file.get(ch)) {
        freqMap[ch]++;
    }

    return freqMap;
}

// 构建Huffman树
HuffmanNode* build(const unordered_map<char, int>& freqMap) {
    priority_queue<HuffmanNode*, vector<HuffmanNode*>, HuffmanNode::Compare> minHeap;

    // 将每个字符及其频率插入到最小堆中
    for (const auto& entry : freqMap) {
        minHeap.push(new HuffmanNode(entry.first, entry.second));
    }

    // 构建Huffman树
    while (minHeap.size() > 1) {
        HuffmanNode* left = minHeap.top();
        minHeap.pop();
        HuffmanNode* right = minHeap.top();
        minHeap.pop();

        HuffmanNode* newNode = new HuffmanNode('\0', left->frequency + right->frequency);
        newNode->left = left;
        newNode->right = right;

        minHeap.push(newNode);
    }

    return minHeap.top(); // 返回根节点
}

// 将Huffman编码表写入文件
void writeHuffmanCodes(const unordered_map<char, string>& huffmanCodes, const string& filename)
    ofstream file(filename);
    for (const auto& entry : huffmanCodes) {
        file << entry.first << ": " << entry.second << endl;
    }
```

```cpp
    file.close();
}

// 将文章编码为二进制字符串
string encodeText(const string& text, const unordered_map<char, string>& huffmanCodes) {
    string encodedText = "";
    for (char ch : text) {
        encodedText += huffmanCodes.at(ch);
    }
    return encodedText;
}

// 将二进制字符串写入文件
void writeb(const string& binaryData, const string& filename) {
    ofstream file(filename, ios::binary);

    // 写入二进制数据
    size_t dataSize = binaryData.size();
    for (size_t i = 0; i < dataSize; i += 8) {
        bitset<8> byte(binaryData.substr(i, 8));
        char byteChar = static_cast<char>(byte.to_ulong());
        file.write(&byteChar, sizeof(byteChar));
    }

    file.close();
}

// 解码函数，通过Huffman树解码二进制文件
string decodeText(const string& binaryData, HuffmanNode* root) {
    string decodedText = "";
    HuffmanNode* currentNode = root;
    for (char bit : binaryData) {
        if (bit == '0') {
            currentNode = currentNode->left;
        } else {
            currentNode = currentNode->right;
        }

        // 如果到达叶子节点，则记录字符并返回根节点
        if (currentNode->left == nullptr && currentNode->right == nullptr) {
            decodedText += currentNode->ch;
            currentNode = root;
        }
```

```
    }
    return decodedText;
}

// 从二进制文件读取数据
string readB(const string& filename) {
    ifstream file(filename, ios::binary);
    string binaryData = "";
    char byte;

    while (file.read(&byte, sizeof(byte))) {
        bitset<8> bits(byte);
        binaryData += bits.to_string();
    }

    return binaryData;
}

int main() {
    // 1. 读取文件并统计字符频率
    string filename = "source.txt";  // 输入文件名
    unordered_map<char, int> freqMap = calculate(filename);

    // 2. 构建Huffman树
    HuffmanNode* root = build(freqMap);

    // 3. 生成Huffman编码
    unordered_map<char, string> huffmanCodes;
    generate(root, "", huffmanCodes);

    // 4. 将Huffman编码表写入文件
    writeHuffmanCodes(huffmanCodes, "Huffman.txt");

    // 5. 读取文件内容并进行Huffman编码
    ifstream file(filename);
    string text((istreambuf_iterator<char>(file)), istreambuf_iterator<char>());
    string encodedText = encodeText(text, huffmanCodes);

    // 6. 将编码结果（以二进制形式）写入code.dat文件
    writeb(encodedText, "code.dat");

    // 7. 读取二进制数据并解码
    string binaryData = readB("code.dat");
```

```
    string decodedText = decodeText(binaryData, root);

    // 8. 将解码后的文本保存到recode.txt
    ofstream recodeFile("recode.txt");
    recodeFile << decodedText;
    recodeFile.close();

    cout << "Huffman encoding and decoding process completed!" << endl;
    return 0;
}
```

6.地铁修建

题目:

给定地铁每段工程的修建时间,求修建整条地铁最少需要多少天。

数据结构:

利用**邻接表**来存储各个节点信息,**W数组**来存储各个地铁段修建时间.

```
void add(int a, int b, int c) {
    e[idx] = b;
    w[idx] = c;
    ne[idx] = h[a];
    h[a] = idx++;
}
```

算法思想:

利用dfs来模拟修建地铁的过程,不断进行回溯,利用**max**和**min**来维护当前修建时间最短的最大地铁段时间.

测试结果:

```
6 6
1 2 4
1 2 4
1 2 4
2 3 4
3 6 7
1 4 2
4 5 5
5 6 6
6
```

源代码:

```
#include<iostream>
#include<algorithm>
#include<cstring>
using namespace std;

#define N 100
int h[N], e[N], ne[N], w[N], idx;
int n, m;
bool visited[N];
 // 记录节点是否被访问过
int minTime = INT_MAX;
// 用来记录最小的最大施工时间
void add(int a, int b, int c) {
    e[idx] = b;
    w[idx] = c;
    ne[idx] = h[a];
    h[a] = idx++;
}

// DFS 深度优先搜索
void dfs(int node, int currentMaxTime) {
    if (node == n) {
        minTime = min(minTime, currentMaxTime);
        return;
    }

    // 遍历当前节点的所有邻接节点
    for (int i = h[node]; i != -1; i = ne[i]) {
        int neighbor = e[i];
        int time = w[i];

        if (!visited[neighbor]) {
            visited[neighbor] = true;

            // 更新当前路径中的最大施工时间
            int newMaxTime = max(currentMaxTime, time);

            // 继续搜索
            dfs(neighbor, newMaxTime);

            visited[neighbor] = false;  // 回溯，撤销访问
```

```
        }
    }
}

int main() {
    memset(h, -1, sizeof(h));  // 初始化邻接表为空
    memset(visited, false, sizeof(visited));  // 初始化访问标记为false

    // 输入交通枢纽和隧道数目
    cin >> n >> m;

    // 输入所有的隧道信息
    for (int i = 0; i < m; i++) {
        int a, b, c;
        cin >> a >> b >> c;
        add(a, b, c);
        add(b, a, c);
    }


    visited[1] = true;
    dfs(1, 0);  // 从1号开始搜索，最大施工时间初始为0

    // 输出最小的最大施工时间
    cout << minTime << endl;

    return 0;
}
```

7.公交线路提示

题目:

输入任意两站点，给出转车次数最少的乘车路线和经过站点最少的乘车路线

数据结构:

建立三个结构体分别存储节点信息,节点的相邻站点信息,线路信息.使用**visited**哈希表记录每个站点在特定线路下的最少转车次数，避免重复访问。通过队列**queue**保存路线.

```cpp
struct Connection {
    int stationId;      // 目标站点ID
    int lineId;         // 所属线路ID
};

// 定义站点结构体
struct Station {
    int id;                             // 站点ID
    unordered_set<int> lines;           // 所属线路ID集合
    vector<Connection> connections;     // 相邻站点的连接信息
};

struct TransferState {
    int stationId;
    int currentLineId;
    int transfers;
    vector<int> path; // 路径记录
};
```

算法思想:

利用**bfs**来求出转车次数最少的乘车路线和经过站点最少的乘车路线.

测试结果:

```
请选择操作:
1. 输入任意两站点，给出转车次数最少的乘车路线。
2. 输入任意两站点，给出经过站点最少的乘车路线。
3. 退出。
请输入选项(1/2/3): 1
请输入起始站点ID: 750
请输入终点站点ID: 2993
转车次数最少的路线 (9 次转车): 750 -> 2162 -> 3291 -> 712 -> 6066 -> 2341 -> 1420 -> 248 -> 4192

请选择操作:
1. 输入任意两站点，给出转车次数最少的乘车路线。
2. 输入任意两站点，给出经过站点最少的乘车路线。
3. 退出。
请输入选项(1/2/3): 2
请输入起始站点ID: 750
请输入终点站点ID: 2993
经过站点最少的路线 (9 个站点): 750 -> 2162 -> 3291 -> 712 -> 6066 -> 2341 -> 1420 -> 248 -> 4192

请选择操作:
1. 输入任意两站点，给出转车次数最少的乘车路线。
2. 输入任意两站点，给出经过站点最少的乘车路线。
3. 退出。
请输入选项(1/2/3): 3
退出程序。
```

源代码:

```cpp
#include <iostream>
#include <vector>
#include <memory>
#include <fstream>
#include <sstream>
#include <unordered_map>
#include <unordered_set>
#include <queue>
#include <tuple>
#include <algorithm>

using namespace std;


struct Connection {
    int stationId;      // 目标站点ID
    int lineId;         // 所属线路ID
};

// 定义站点结构体
struct Station {
    int id;                             // 站点ID
    unordered_set<int> lines;           // 所属线路ID集合
    vector<Connection> connections;     // 相邻站点的连接信息
};

struct TransferState {
    int stationId;
    int currentLineId;
    int transfers;
    vector<int> path; // 路径记录
};
/**
 *  从文件中读取公交线路数据并构建线路图
 *
 *  filename 数据文件名
 *  stations 存储所有站点的映射（站点ID -> Station）
 *  true 读取成功
 *  false 读取失败
 */
bool readBusData(const string& filename, unordered_map<int, Station>& stations) {
```

```cpp
// 打开数据文件
ifstream infile(filename);
if (!infile.is_open()) {
    cerr << "无法打开文件 " << filename << endl;
    return false;
}

string line;
// 记录每条线路的最后一个站点ID
unordered_map<int, int> lastStationPerLine;

// 读取文件中的每一行
while (getline(infile, line)) {
    // 跳过标题行
    if (line.find("公交线路ID") != string::npos) {
        continue;
    }

    // 使用字符串流解析每一行的数据
    stringstream ss(line);
    int lineId, stationOrder, stationId;
    if (!(ss >> lineId >> stationOrder >> stationId)) {
        cerr << "解析错误: " << line << endl;
        continue;  // 跳过解析失败的行
    }

    // 获取当前站点，如果不存在则创建
    if (stations.find(stationId) == stations.end()) {
        stations[stationId] = Station{stationId, {}, {}};
    }
    // 添加所属线路
    stations[stationId].lines.insert(lineId);

    // 如果不是起始站点，建立与前一个站点的连接
    if (stationOrder > 0) {
        if (lastStationPerLine.find(lineId) != lastStationPerLine.end()) {
            int lastStationId = lastStationPerLine[lineId];
            // 建立双向连接
            stations[lastStationId].connections.push_back(Connection{stationId, lineId});
            stations[stationId].connections.push_back(Connection{lastStationId, lineId});
        }
        // 更新最后一个站点
        lastStationPerLine[lineId] = stationId;
```

```cpp
        } else {
            // 起始站点，更新最后一个站点
            lastStationPerLine[lineId] = stationId;
        }
    }

    infile.close();
    return true;
}



/**
 *  查找转车次数最少的路线
 *
 *  startId 起始站点ID
 *  endId 终点站点ID
 *  stations 站点图
 *  resultPath 结果路径
 *  true 找到路径
 *  false 未找到路径
 */
bool findMinTransfersPath(int startId, int endId, const unordered_map<int, Station>& stations, \
    if (stations.find(startId) == stations.end() || stations.find(endId) == stations.end()) {
        cerr << "起始站点或终点站点不存在。" << endl;
        return false;
    }

    // 使用队列进行BFS
    queue<TransferState> q;

    // 访问标记：stationId -> lineId -> 最小转车次数
    unordered_map<int, unordered_map<int, int>> visited;

    // 初始化队列：从起始站点出发，可以选择任何一条线路
    for (const auto& line : stations.at(startId).lines) {
        TransferState state;
        state.stationId = startId;
        state.currentLineId = line;
        state.transfers = 0;
        state.path.push_back(startId);
        q.push(state);
        visited[startId][line] = 0;
    }
```

```cpp
while (!q.empty()) {
    TransferState current = q.front();
    q.pop();

    // 如果到达终点，记录路径
    if (current.stationId == endId) {
        resultPath = current.path;
        return true;
    }

    // 遍历相邻站点
    auto it = stations.find(current.stationId);
    if (it == stations.end()) continue;

    for (const auto& conn : it->second.connections) {
        int neighborId = conn.stationId;
        int lineId = conn.lineId;

        // 判断是否需要转车
        int newTransfers = current.transfers;
        if (lineId != current.currentLineId) {
            newTransfers += 1;
        }

        // 检查是否已经访问过，或者是否有更少的转车次数
        if (visited.find(neighborId) != visited.end() &&
            visited.at(neighborId).find(lineId) != visited.at(neighborId).end() &&
            visited.at(neighborId).at(lineId) <= newTransfers) {
            continue;
        }

        // 标记为已访问
        visited[neighborId][lineId] = newTransfers;

        // 记录路径
        TransferState nextState;
        nextState.stationId = neighborId;
        nextState.currentLineId = lineId;
        nextState.transfers = newTransfers;
        nextState.path = current.path;
        nextState.path.push_back(neighborId);
        q.push(nextState);
```

```
        }
    }

    // 如果没有找到路径
    return false;
}

/**
 *  查找经过站点最少的路线
 *
 *  startId 起始站点ID
 *  endId 终点站点ID
 *  stations 站点图
 *  resultPath 结果路径
 *  true 找到路径
 *  false 未找到路径
 */
bool findMinStopsPath(int startId, int endId, const unordered_map<int, Station>& stations, vect
    if (stations.find(startId) == stations.end() || stations.find(endId) == stations.end()) {
        cerr << "起始站点或终点站点不存在。" << endl;
        return false;
    }

    // 使用队列进行BFS
    queue<vector<int>> q;
    unordered_set<int> visited;
    visited.insert(startId);

    // 初始化队列
    q.push({startId});

    while (!q.empty()) {
        vector<int> path = q.front();
        q.pop();

        int currentStation = path.back();

        // 如果到达终点，记录路径
        if (currentStation == endId) {
            resultPath = path;
            return true;
        }
```

```cpp
        // 遍历相邻站点
        auto it = stations.find(currentStation);
        if (it == stations.end()) continue;

        for (const auto& conn : it->second.connections) {
            int neighborId = conn.stationId;
            if (visited.find(neighborId) == visited.end()) {
                visited.insert(neighborId);
                vector<int> newPath = path;
                newPath.push_back(neighborId);
                q.push(newPath);
            }
        }
    }

    // 如果没有找到路径
    return false;
}

int main() {
    // 创建一个存储所有站点的映射
    unordered_map<int, Station> stations;

    // 调用函数读取数据文件
    string filename = "road_test.txt";
    if (!readBusData(filename, stations)) {
        return 1;  // 如果读取失败，退出程序
    }

    // 用户交互
    while (true) {
        cout << "\n请选择操作:\n";
        cout << "1. 输入任意两站点，给出转车次数最少的乘车路线。\n";
        cout << "2. 输入任意两站点，给出经过站点最少的乘车路线。\n";
        cout << "3. 退出。\n";
        cout << "请输入选项(1/2/3): ";
        int choice;
        cin >> choice;

        if (choice == 3) {
            cout << "退出程序。" << endl;
            break;
        }
```

```
        if (choice != 1 && choice != 2) {
            cout << "无效的选项，请重新选择。" << endl;
            continue;
        }

        int startId, endId;
        cout << "请输入起始站点ID: ";
        cin >> startId;
        cout << "请输入终点站点ID: ";
        cin >> endId;

        vector<int> path;
        if (choice == 1) {
            // 最少转车次数
            if (findMinTransfersPath(startId, endId, stations, path)) {
                cout << "转车次数最少的路线 (" << path.size() - 1 << " 次转车): ";
                for (size_t i = 0; i < path.size(); ++i) {
                    cout << path[i];
                    if (i != path.size() - 1) cout << " -> ";
                }
                cout << endl;
            } else {
                cout << "未找到从站点 " << startId << " 到站点 " << endId << " 的路线。" << endl;
            }
        } else if (choice == 2) {
            // 最少经过站点
            if (findMinStopsPath(startId, endId, stations, path)) {
                cout << "经过站点最少的路线 (" << path.size() - 1 << " 个站点): ";
                for (size_t i = 0; i < path.size(); ++i) {
                    cout << path[i];
                    if (i != path.size() - 1) cout << " -> ";
                }
                cout << endl;
            } else {
                cout << "未找到从站点 " << startId << " 到站点 " << endId << " 的路线。" << endl;
            }
        }
    }

    return 0;
}
```

8.B-Tree

题目:

对1-10000的所有质数，建立m=4的B-tree（每个非叶子结点至少包含1个关键字即2棵子树，最多3个关键字即4棵子树）。

数据结构:

使用**Node**结构体来存储各个节点,并且使用**vector**来存储各个节点的关键字,子节点,再存储父节点和是否为**leaf**.

```
struct Node {
    vector<int> data;
    vector<Node*> next;
    Node* parent;
    bool leaf = true;

    Node() {
        parent = nullptr;
    }
};
```

算法思想:

两种判断,第一种如果插入节点以后超过m-1,则进行分裂,第二种如果插入节点以后小于等于m-1,则进行合并.分裂即将mid=(m-1)/2+1作为新的根节点,再进行分裂,不断递归.合并即删除根节点,再进行合并.

测试结果:

```
211 yes
223 yes
227 yes
229 yes
233 yes
239 yes
241 yes
251 yes
257 yes
263 yes
269 yes
271 yes
277 yes
281 yes
283 yes
293 yes
```

```
601 no
607 no
613 no
617 no
619 no
631 no
641 no
643 no
647 no
653 no
659 no
661 no
673 no
677 no
683 no
691 no
```

100 yes
102 yes
104 yes
106 yes
108 yes
110 yes
112 yes
114 yes
116 yes
118 yes
120 yes
122 yes
124 yes
126 yes
128 yes
130 yes
132 yes
134 yes
136 yes
138 yes
140 yes
142 yes
144 yes
146 yes
148 yes
150 yes
152 yes
154 yes
156 yes
158 yes
160 yes
162 yes
164 yes
166 yes
168 yes
170 yes
172 yes
174 yes
176 yes
178 yes
180 yes
182 yes
184 yes

186 yes
188 yes
190 yes
192 yes
194 yes
196 yes
198 yes
200 yes

源代码:

```cpp
#include <iostream>
#include <vector>
#include <fstream>
#include <cmath>
#define order 4  // 阶数
using namespace std;
struct Node;
void remove(Node*& root, int key);
void remove(Node*& root, Node* node, int key);
void rebalance(Node*& root, Node* node);

struct Node {
    vector<int> data;
    vector<Node*> next;
    Node* parent;
    bool leaf = true;

    Node() {
        parent = nullptr;
    }
};

// 判断是否为质数
bool isPrime(int n) {
    if (n < 2) return false;
    for (int i = 2; i <= sqrt(n); i++) {
        if (n % i == 0) return false;
    }
    return true;
}

// 查找函数
Node* Find(Node* root, int x) {
    if (!root) return nullptr;

    int i = 0;
    while (i < root->data.size() && root->data[i] < x) {
        i++;
    }

    if (i < root->data.size() && root->data[i] == x) {
```

```cpp
        return root;
    }

    if (root->leaf) {
        return nullptr;
    }

    return Find(root->next[i], x);
}

// 分裂节点
void split(Node*& root, Node* node) {
    int mid = node->data.size() / 2;
    int mid_value = node->data[mid];

    Node* right = new Node();
    right->leaf = node->leaf;

    // 移动数据到右节点
    for (int i = mid + 1; i < node->data.size(); i++) {
        right->data.push_back(node->data[i]);
    }

    // 移动子节点指针
    if (!node->leaf) {
        for (int i = mid + 1; i <= node->data.size(); i++) {
            right->next.push_back(node->next[i]);
            node->next[i]->parent = right;
        }
    }

    // 调整原节点
    node->data.resize(mid);
    if (!node->leaf) {
        node->next.resize(mid + 1);
    }

    if (node->parent == nullptr) {
        // 创建新根
        Node* new_root = new Node();
        new_root->leaf = false;
        new_root->data.push_back(mid_value);
        new_root->next.push_back(node);
```

```cpp
        new_root->next.push_back(right);
        node->parent = new_root;
        right->parent = new_root;
        root = new_root;
    } else {
        // 插入到父节点
        Node* parent = node->parent;
        int pos = 0;
        while (pos < parent->data.size() && parent->data[pos] < mid_value) {
            pos++;
        }
        parent->data.insert(parent->data.begin() + pos, mid_value);
        parent->next.insert(parent->next.begin() + pos + 1, right);
        right->parent = parent;

        if (parent->data.size() > order - 1) {
            split(root, parent);
        }
    }
}

// 插入函数
void insert(Node*& root, int x) {
    if (!root) {
        root = new Node();
        root->data.push_back(x);
        return;
    }

    Node* node = root;
    while (!node->leaf) {
        int i = 0;
        while (i < node->data.size() && node->data[i] < x) {
            i++;
        }
        node = node->next[i];
    }

    // 插入到叶子节点
    int i = 0;
    while (i < node->data.size() && node->data[i] < x) {
        i++;
    }
```

```
    if (i < node->data.size() && node->data[i] == x) {
        return;
    }

    node->data.insert(node->data.begin() + i, x);

    if (node->data.size() > order - 1) {
        split(root, node);
    }
}

// 查找最小值
int findMin(Node* root) {
    if (!root) return -1;
    while (!root->leaf) {
        root = root->next[0];
    }
    return root->data[0];
}

// 查找最大值
int findMax(Node* root) {
    if (!root) return -1;
    while (!root->leaf) {
        root = root->next[root->next.size() - 1];
    }
    return root->data[root->data.size() - 1];
}

// 合并节点
void merge(Node*& root, Node* parent, int index) {
    Node* left = parent->next[index];
    Node* right = parent->next[index + 1];

    // 将父节点的关键字下移
    left->data.push_back(parent->data[index]);

    // 将右兄弟的关键字和子节点合并到左兄弟
    for (int i = 0; i < right->data.size(); i++) {
        left->data.push_back(right->data[i]);
    }

    if (!right->leaf) {
```

```
        for (int i = 0; i < right->next.size(); i++) {
            left->next.push_back(right->next[i]);
            right->next[i]->parent = left;
        }
    }

    // 从父节点删除关键字和指针
    parent->data.erase(parent->data.begin() + index);
    parent->next.erase(parent->next.begin() + index + 1);

    delete right;

    // 如果父节点是根节点且为空，更新根
    if (parent == root && parent->data.empty()) {
        root = left;
        root->parent = nullptr;
        delete parent;
    }
    // 如果父节点关键字太少，需要重新平衡
    else if (parent != root && parent->data.size() < (order - 1) / 2) {
        rebalance(root, parent);
    }
}

// 从兄弟节点借一个关键字
void borrowFromSibling(Node* node, int index, bool fromLeft) {
    Node* parent = node->parent;

    if (fromLeft) {
        Node* leftSibling = parent->next[index - 1];

        // 将父节点的关键字下移到当前节点
        node->data.insert(node->data.begin(), parent->data[index - 1]);

        // 将左兄弟的最大关键字上移到父节点
        parent->data[index - 1] = leftSibling->data.back();
        leftSibling->data.pop_back();

        if (!node->leaf) {
            // 移动对应的子节点
            node->next.insert(node->next.begin(), leftSibling->next.back());
            leftSibling->next.back()->parent = node;
            leftSibling->next.pop_back();
```

```
        }
    } else {
        Node* rightSibling = parent->next[index + 1];

        // 将父节点的关键字下移到当前节点
        node->data.push_back(parent->data[index]);

        // 将右兄弟的最小关键字上移到父节点
        parent->data[index] = rightSibling->data.front();
        rightSibling->data.erase(rightSibling->data.begin());

        if (!node->leaf) {
            // 移动对应的子节点
            node->next.push_back(rightSibling->next.front());
            rightSibling->next.front()->parent = node;
            rightSibling->next.erase(rightSibling->next.begin());
        }
    }
}

// 重新平衡节点
void rebalance(Node*& root, Node* node) {
    if (node == root) return;

    Node* parent = node->parent;
    int index = 0;
    while (index < parent->next.size() && parent->next[index] != node) {
        index++;
    }

    // 尝试从左兄弟借
    if (index > 0) {
        Node* leftSibling = parent->next[index - 1];
        if (leftSibling->data.size() > (order - 1) / 2) {
            borrowFromSibling(node, index, true);
            return;
        }
    }

    // 尝试从右兄弟借
    if (index < parent->next.size() - 1) {
        Node* rightSibling = parent->next[index + 1];
        if (rightSibling->data.size() > (order - 1) / 2) {
```

```cpp
            borrowFromSibling(node, index, false);
            return;
        }
    }

    // 如果无法借，则需要合并
    if (index > 0) {
        merge(root, parent, index - 1);
    } else {
        merge(root, parent, index);
    }
}

// 从叶子节点删除关键字
void deleteFromLeaf(Node*& root, Node* node, int key) {
    int index = 0;
    while (index < node->data.size() && node->data[index] < key) {
        index++;
    }

    if (index < node->data.size() && node->data[index] == key) {
        node->data.erase(node->data.begin() + index);

        if (node == root) {
            if (node->data.empty()) {
                delete root;
                root = nullptr;
            }
        }
        else if (node->data.size() < (order - 1) / 2) {
            rebalance(root, node);
        }
    }
}

// 从内部节点删除关键字
void deleteFromInternal(Node*& root, Node* node, int key) {
    int index = 0;
    while (index < node->data.size() && node->data[index] < key) {
        index++;
    }

    if (index < node->data.size() && node->data[index] == key) {
```

```
        if (node->next[index]->data.size() >= (order + 1) / 2) {
            // 使用前驱
            int pred = findMax(node->next[index]);
            node->data[index] = pred;
            remove(root, node->next[index], pred);
        } else if (node->next[index + 1]->data.size() >= (order + 1) / 2) {
            // 使用后继
            int succ = findMin(node->next[index + 1]);
            node->data[index] = succ;
            remove(root, node->next[index + 1], succ);
        } else {
            // 合并节点
            merge(root, node, index);
            remove(root, key);
        }
    } else if (!node->leaf) {
        remove(root, node->next[index], key);
    }
}

// 删除函数
void remove(Node*& root, Node* node, int key) {
    if (!node) return;

    int index = 0;
    while (index < node->data.size() && node->data[index] < key) {
        index++;
    }

    if (node->leaf) {
        deleteFromLeaf(root, node, key);
    } else {
        if (index < node->data.size() && node->data[index] == key) {
            deleteFromInternal(root, node, key);
        } else {
            remove(root, node->next[index], key);
        }
    }
}

// 删除入口函数
void remove(Node*& root, int key) {
    if (!root) return;
```

```cpp
    remove(root, root, key);
}

// 将查询结果写入文件
void writeResult(const string& filename, int num, bool found) {
    ofstream out(filename, ios::app);
    out << num << (found ? " yes" : " no") << endl;
    out.close();
}

int main() {
    Node* root = nullptr;

    // 构建初始B树（插入1-10000的所有质数）
    for (int i = 1; i <= 10000; i++) {
        if (isPrime(i)) {
            insert(root, i);
        }
    }

    // 任务1：查询200-300的每个数
    ofstream out1("b-tree1.txt");
    out1.close();  // 清空文件
    for (int i = 200; i <= 300; i++) {
        bool found = (Find(root, i) != nullptr);
        writeResult("b-tree1.txt", i, found);
    }

    // 任务2：删除500-2000中的质数并查询600-700的质数
    for (int i = 500; i <= 2000; i++) {
        if (isPrime(i)) {
            remove(root, i);
        }
    }

    ofstream out2("b-tree2.txt");
    out2.close();
    for (int i = 600; i <= 700; i++) {
        if (isPrime(i)) {
            bool found = (Find(root, i) != nullptr);
            writeResult("b-tree2.txt", i, found);
        }
    }
```

```
    // 任务3：插入1-1000的偶数并查询100-200的偶数
    for (int i = 2; i <= 1000; i += 2) {
        insert(root, i);
    }

    ofstream out3("b-tree3.txt");
    out3.close();
    for (int i = 100; i <= 200; i += 2) {
        bool found = (Find(root, i) != nullptr);
        writeResult("b-tree3.txt", i, found);
    }

    return 0;
}
```

9.排序比较

题目：

利用随机函数产生10个样本，每个样本有100000个随机整数（并使第一个样本是正序，第二个样本是逆序），利用直接插入排序、希尔排序，冒泡排序、快速排序、选择排序、堆排序，归并排序、基数排序8种排序方法进行排序

数据结构：

利用**vector**来存储各个样本.

算法思想：

1. 插入排序 (Insertion Sort)

- **思想**：将待排序元素逐个插入到已经排好序的部分中。每次插入时，保持左边部分是有序的。
- **过程**：从第二个元素开始，逐个与前面已排好序的元素进行比较并插入合适位置。

2. 希尔排序 (Shell Sort)

- **思想**：插入排序的改进版，采用分组的方式减少数据移动的次数。通过逐步减小间隔来提高插入排序的效率。
- **过程**：首先按一定间隔将数据分成多个子序列，每个子序列独立进行插入排序，然后逐步减小间隔，直到间隔为 1（即进行普通插入排序）。

3. 冒泡排序 (Bubble Sort)

- **思想**：通过多次交换相邻元素的方式，将最大（或最小）元素逐渐“冒泡”到序列的一端。
- **过程**：依次比较相邻元素，若顺序错误则交换，直到没有任何交换发生为止。

4. 快速排序 (Quick Sort)

- **思想**：通过分治法（Divide and Conquer）将数据分为两部分，然后递归地对两部分进行排序。每次选择一个“基准”元素，将比基准小的元素放左边，比基准大的元素放右边。
- **过程**：选定一个基准元素，将数组分成两部分，对两部分递归排序。

5. 选择排序 (Selection Sort)

- **思想**：每次从未排序的部分选择最小（或最大）元素，并将其放到已排序部分的末尾。
- **过程**：通过选择最小（或最大）元素交换到当前未排序部分的最前面。

6. 堆排序 (Heap Sort)

- **思想**：利用堆（特别是二叉堆）数据结构来进行排序。将待排序数组转化为一个最大堆或最小堆，通过不断提取堆顶元素来完成排序。
- **过程**：首先构建最大堆，然后依次交换堆顶元素与末尾元素，并重新调整堆结构。

7. 归并排序 (Merge Sort)

- **思想**：通过分治法将数组分成两个子数组，分别对这两个子数组递归排序，然后合并两个已排序的子数组。
- **过程**：将数组不断分割成两部分，对每部分递归排序，最后合并已排序的部分。

8. 基数排序 (Radix Sort)

- **思想**：通过逐位排序来实现整体的排序。基于数字的每一位进行排序，通常使用计数排序作为稳定的子排序算法。
- **过程**：从最低位（或最高位）开始，依次对每一位进行排序，直到所有位数都排序完。

测试结果(部分):

```
Testing data0.txt
Insertion Sort took 5744 ms
Shell Sort took 20 ms
Bubble Sort took 31428 ms
Quick Sort took 9 ms
Selection Sort took 10568 ms
Heap Sort took 33 ms
Merge Sort took 13 ms
Radix Sort took 8 ms
```

源代码:

```cpp
#include<iostream>
#include<vector>
#include<queue>
#include<ctime>
#include<fstream>
#include <cstdlib>
#include <string>
#include <chrono>  // 需要添加头文件以使用时间测量
using namespace std;

void insert(vector<int> &p) {
    for (int i = 1; i < p.size(); i++) {
        int temp = p[i];
        int j = i - 1;
        while (j >= 0 && p[j] > temp) {
            p[j + 1] = p[j];
            j--;
        }
        p[j + 1] = temp;
    }
}

void shellSort(vector<int> &p) {
    int n = p.size();
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = p[i];
            int j = i;
            while (j >= gap && p[j - gap] > temp) {
                p[j] = p[j - gap];
                j -= gap;
            }
            p[j] = temp;
        }
    }
}

void bubblesort(vector<int> &p) {
    for (int i = 0; i < p.size(); i++) {
        for (int j = 0; j < p.size() - i - 1; j++) {
            if (p[j] > p[j + 1]) {
```

```cpp
                swap(p[j], p[j + 1]);
            }
        }
    }
}

void quicksort(vector<int> &p, int l, int r) {
    if (l >= r) return;
    int i = l, j = r;
    int temp = p[l];
    while (i < j) {
        while (i < j && p[j] >= temp) j--;
        p[i] = p[j];
        while (i < j && p[i] <= temp) i++;
        p[j] = p[i];
    }
    p[i] = temp;
    quicksort(p, l, i - 1);
    quicksort(p, i + 1, r);
}

void selectsort(vector<int> &p) {
    for (int i = 0; i < p.size(); i++) {
        int minIndex = i;
        for (int j = i + 1; j < p.size(); j++) {
            if (p[j] < p[minIndex]) {
                minIndex = j;
            }
        }
        swap(p[i], p[minIndex]);
    }
}

void heapSort(vector<int>& arr) {
    priority_queue<int, vector<int>, greater<int>> pq;  // 使用最小堆
    for (int num : arr) {
        pq.push(num);
    }
    int idx = 0;
    while (!pq.empty()) {
        arr[idx++] = pq.top();
        pq.pop();
    }
```

```
}

void merge(vector<int> &p, vector<int> &t, int l, int r) {
    if (l >= r) {
        return;
    }
    int mid = (l + r) / 2;
    merge(p, t, l, mid);
    merge(p, t, mid + 1, r);
    int k = 0;
    int i = l;
    int j = mid + 1;
    while (i <= mid && j <= r) {
        if (p[i] <= p[j]) {
            t[k++] = p[i++];
        } else {
            t[k++] = p[j++];
        }
    }
    while (i <= mid) {
        t[k++] = p[i++];
    }
    while (j <= r) {
        t[k++] = p[j++];
    }
    k = 0;
    for (int i = l; i <= r; i++) {
        p[i] = t[k++];
    }
}

int cal(int x) {
    int i = 0;
    while (x > 0) {
        x /= 10;
        i++;
    }
    return i;  // 返回位数
}

void radix(vector<int> &p) {
    int max1 = 0;
    for (int i = 0; i < p.size(); i++) {
```

```cpp
            max1 = max(max1, p[i]);
        }
        max1 = cal(max1);
        vector<vector<int>> q(10);
        int r = 1;
        for (int i = 0; i < max1; i++) {
            for (int j = 0; j < p.size(); j++) {
                int k = p[j] / r % 10;
                q[k].push_back(p[j]);
            }
            int k = 0;
            for (int j = 0; j < 10; j++) {
                for (int l = 0; l < q[j].size(); l++) {
                    p[k++] = q[j][l];
                }
            }
            for (int j = 0; j < 10; j++) {
                q[j].clear();
            }
            r *= 10;
        }
    }

    // 从文件读取数据
    void loadDataFromFile(const string &file, vector<int> &p) {
        ifstream inFile(file);
        int num;
        while (inFile >> num) {
            p.push_back(num);
        }
    }

    // 测试排序算法
    void testSortingAlgorithms(const string &file) {
        vector<int> data;
        loadDataFromFile(file, data);  // 从文件加载数据

        // 测试所有排序算法
        vector<int> dataCopy;

        // 插入排序
        dataCopy = data;
        auto start = chrono::high_resolution_clock::now();
```

```cpp
insert(dataCopy);
auto stop = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(stop - start);
cout << "Insertion Sort took " << duration.count() << " ms" << endl;

// 希尔排序
dataCopy = data;
start = chrono::high_resolution_clock::now();
shellSort(dataCopy);
stop = chrono::high_resolution_clock::now();
duration = chrono::duration_cast<chrono::milliseconds>(stop - start);
cout << "Shell Sort took " << duration.count() << " ms" << endl;

// 冒泡排序
dataCopy = data;
start = chrono::high_resolution_clock::now();
bubblesort(dataCopy);
stop = chrono::high_resolution_clock::now();
duration = chrono::duration_cast<chrono::milliseconds>(stop - start);
cout << "Bubble Sort took " << duration.count() << " ms" << endl;

// 快速排序
dataCopy = data;
start = chrono::high_resolution_clock::now();
quicksort(dataCopy, 0, dataCopy.size() - 1);
stop = chrono::high_resolution_clock::now();
duration = chrono::duration_cast<chrono::milliseconds>(stop - start);
cout << "Quick Sort took " << duration.count() << " ms" << endl;

// 选择排序
dataCopy = data;
start = chrono::high_resolution_clock::now();
selectsort(dataCopy);
stop = chrono::high_resolution_clock::now();
duration = chrono::duration_cast<chrono::milliseconds>(stop - start);
cout << "Selection Sort took " << duration.count() << " ms" << endl;

// 堆排序
dataCopy = data;
start = chrono::high_resolution_clock::now();
heapSort(dataCopy);
stop = chrono::high_resolution_clock::now();
duration = chrono::duration_cast<chrono::milliseconds>(stop - start);
```

```
    cout << "Heap Sort took " << duration.count() << " ms" << endl;

    // 归并排序
    dataCopy = data;
    vector<int> temp(dataCopy.size());
    start = chrono::high_resolution_clock::now();
    merge(dataCopy, temp, 0, dataCopy.size() - 1);
    stop = chrono::high_resolution_clock::now();
    duration = chrono::duration_cast<chrono::milliseconds>(stop - start);
    cout << "Merge Sort took " << duration.count() << " ms" << endl;

    // 基数排序
    dataCopy = data;
    start = chrono::high_resolution_clock::now();
    radix(dataCopy);
    stop = chrono::high_resolution_clock::now();
    duration = chrono::duration_cast<chrono::milliseconds>(stop - start);
    cout << "Radix Sort took " << duration.count() << " ms" << endl;
}
void generateR(vector<int> &p,string file){
    srand(time(0));
    p.clear();
    for(int i=0;i<100000;i++){
        p.push_back(rand()%1000000);
    }
    ofstream outFile(file);
    for (const int &num : p) {
        outFile << num << endl;  // 每个数字占一行
    }
}
int main() {
    vector<int> p;
    generateR(p,"data0.txt");
    for(int i:p){
        a[i]++;
    }
    for(int i=0;i<100000;i++){
        if(a[i]!=0){
            cout<<i<<":"<<a[i]<<endl;
        }
    }
    for (char i = '0'; i <= '9'; i++) {
        string file = "data" + string(1, i) + ".txt";
```

```
        cout << "Testing " << file << endl;
        generateR(p,file);
        testSortingAlgorithms(file);  // 对每个文件进行排序测试
    }
    return 0;
}
```

选做题

1. 最小生成树

题目:

利用普利姆算法和克鲁斯卡尔算法实现最小生成树问题。

数据结构:

使用三元组存储顶点和权重,在使用vector存储各个顶点集合,来进行并查集的实现.

```
struct Edge {
    int u, v;
    float weight;

    // 用于排序
    bool operator<(Edge& other) const {
        return weight < other.weight;
    }
};


int find(vector<int>& parent, int i) {
        if (parent[i] != i) {
            parent[i] = find(parent, parent[i]);
        }
        return parent[i];
    }

    // 并查集合并函数
    void unionSets(vector<int>& parent, int x, int y) {
        parent[find(parent, x)] = find(parent, y);
    }
```

算法思想:

- **prim算法**:先找距离最小的点,然后更新到集合的距离,直到所有点都被访问.
- **kruskal算法**:先对所有边进行排序,然后依次加入最小但不够成回路的边,直到所有点都被访问.

测试数据:

```
6
0 1 4.0
0 2 3.0
1 2 1.0
1 3 2.0
2 3 4.0
3 4 2.0
3 5 6.0
4 5 1.0
```

测试结果:

```
使用Prim算法:
Prim算法生成的最小生成树:
0 - 2 : 3
2 - 1 : 1
1 - 3 : 2
3 - 4 : 2
4 - 5 : 1
最小生成树的总权值:9

使用Kruskal算法:
Kruskal算法生成的最小生成树:
1 - 2 : 1
4 - 5 : 1
1 - 3 : 2
3 - 4 : 2
0 - 2 : 3
最小生成树的总权值:9
```

源代码:

```cpp
#include <iostream>
#include <vector>
#include <fstream>
#include <queue>
#include <climits>
#include <algorithm>
#define MAX 9999999.0
using namespace std;

struct Edge {
    int u, v;
    float weight;

    // 用于排序
    bool operator<(Edge& other) const {
        return weight < other.weight;
    }
};

class Graph {
private:
    int V; // 顶点数
    vector<Edge> edges; // 边集合

    // 并查集查找函数
    int find(vector<int>& parent, int i) {
        if (parent[i] != i) {
            parent[i] = find(parent, parent[i]);
        }
        return parent[i];
    }

    // 并查集合并函数
    void unionSets(vector<int>& parent, int x, int y) {
        parent[find(parent, x)] = find(parent, y);
    }

public:
    // 从文件读取图数据
    Graph(const string& filename) {
        ifstream file(filename);
```

```
    if (!file.is_open()) {
        cerr << "无法打开文件" << filename << endl;
        exit(1);
    }

    // 读取顶点数
    file >> V;

    // 读取边的信息
    int u, v;
    float w;
    while (file >> u >> v >> w) {
        edges.push_back({u, v, w});
    }
    file.close();
}

// Prim算法实现
void primMST() {
    vector<bool> visited(V, false);
    vector<float> key(V, MAX);
    vector<int> parent(V, -1);

    // 从顶点0开始
    key[0] = 0;

    float totalWeight = 0;
    cout << "\nPrim算法生成的最小生成树：" << endl;

    for (int count = 0; count < V; count++) {
        // 找到未访问顶点中key值最小的顶点
        float minKey = MAX;
        int u = -1;

        for (int v = 0; v < V; v++) {
            if (!visited[v] && key[v] < minKey) {
                minKey = key[v];
                u = v;
            }
        }

        if (u == -1) break;
```

```
        visited[u] = true;

        // 如果不是起始顶点，输出这条边
        if (parent[u] != -1) {
            cout << parent[u] << " - " << u << " : " << key[u] << endl;
            totalWeight += key[u];
        }

        // 更新相邻顶点的key值
        for (const Edge& edge : edges) {
            if ((edge.u == u || edge.v == u) && !visited[edge.u == u ? edge.v : edge.u]) {
                int v = edge.u == u ? edge.v : edge.u;
                if (edge.weight < key[v]) {
                    parent[v] = u;
                    key[v] = edge.weight;
                }
            }
        }
    }

    cout << "最小生成树的总权值: " << totalWeight << endl;
}

// Kruskal算法实现
void kruskalMST() {
    vector<Edge> result;
    vector<int> parent(V);

    // 初始化并查集
    for (int i = 0; i < V; i++) {
        parent[i] = i;
    }

    // 对边按权重排序
    sort(edges.begin(), edges.end());

    float totalWeight = 0;
    cout << "\nKruskal算法生成的最小生成树: " << endl;

    for (const Edge& edge : edges) {
        int setU = find(parent, edge.u);
        int setV = find(parent, edge.v);
```

```
            // 如果加入这条边不会形成环
            if (setU != setV) {
                cout << edge.u << " - " << edge.v << " : " << edge.weight << endl;
                totalWeight += edge.weight;
                unionSets(parent, setU, setV);
            }
        }

        cout << "最小生成树的总权值: " << totalWeight << endl;
    }
};

int main() {
    string filename = "graph.txt";
    Graph g(filename);

    cout << "使用Prim算法: ";
    g.primMST();

    cout << "\n使用Kruskal算法: ";
    g.kruskalMST();

    return 0;
}
```

2. 计算表达式

题目:

一个算术表达式是由操作数(operand)、运算符(operator)和界限符(delimiter)组成的。假设操作数是正实数，运算符只含加减乘除等四种运算符，界限符有左右括号和表达式起始、结束符“#”，如：#6+15*（21-8/4）#。引入表达式起始、结束符是为了方便。编程利用“运算符优先法”求算术表达式的值。

数据结构:

使用栈来存储操作数和运算符,使用map来存储运算符的优先级.

```cpp
template <typename T>
class MyVector {
private:
    T* arr;  // 存储数组元素的数组
    int capacity;  // 数组的最大容量
    int size;  // 当前数组的大小
};
```

```cpp
template <typename T>
class MyStack {
private:
    T* arr;  // 存储栈元素的数组
    int capacity;  // 栈的最大容量
    int size;  // 当前栈的大小
};
```

算法思想:

从左到右扫描表达式,遇到操作数入栈,遇到运算符则根据优先级进行运算. 运算符优先级使用map来存储,方便查找.

测试数据:

```
#6+15*（21-8/4）#
```

测试结果:

```
291
```

源代码:

```cpp
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;

// 自定义栈
template <typename T>
class MyStack {
private:
    T* arr;  // 存储栈元素的数组
    int capacity;  // 栈的最大容量
    int size;  // 当前栈的大小

public:
    MyStack(int cap = 100) {  // 构造函数，设置初始容量
        capacity = cap;
        arr = new T[capacity];
        size = 0;
    }

    ~MyStack() {  // 析构函数，释放内存
        delete[] arr;
    }

    void push(const T& value) {
        if (size == capacity) {
            // 栈满时，扩大容量
            capacity *= 2;
            T* newArr = new T[capacity];
            for (int i = 0; i < size; i++) {
                newArr[i] = arr[i];
            }
            delete[] arr;
            arr = newArr;
        }
        arr[size++] = value;
    }

    void pop() {
        if (size > 0) {
            --size;
```

```
        }
    }

    T top() const {
        if (size > 0) {
            return arr[size - 1];
        }
        throw runtime_error("Stack is empty");
    }

    bool empty() const {
        return size == 0;
    }

    int getSize() const {
        return size;
    }
};
// 自定义动态数组
template <typename T>
class MyVector {
private:
    T* arr;  // 存储数组元素的数组
    int capacity;  // 数组的最大容量
    int size;  // 当前数组的大小

public:
    MyVector(int cap = 100) {
        capacity = cap;
        arr = new T[capacity];
        size = 0;
    }

    ~MyVector() {
        delete[] arr;
    }

    void push_back(const T& value) {
        if (size == capacity) {
            // 扩容
            capacity *= 2;
            T* newArr = new T[capacity];
            for (int i = 0; i < size; i++) {
```

```
                newArr[i] = arr[i];
            }
            delete[] arr;
            arr = newArr;
        }
        arr[size++] = value;
    }

    T operator[](int index) const {
        if (index >= 0 && index < size) {
            return arr[index];
        }
        throw runtime_error("Index out of bounds");
    }

    int getSize() const {
        return size;
    }
};
// 优先级表
unordered_map<char, int> h{ {'+', 1}, {'-', 1}, {'*', 2}, {'/', 2} };

// 自定义栈类型
MyStack<int> num;
MyStack<char> op;

void eval() {  // 求值
    int a = num.top();  // 第二个操作数
    num.pop();

    int b = num.top();  // 第一个操作数
    num.pop();

    char p = op.top();  // 运算符
    op.pop();

    int r = 0;  // 结果

    // 计算结果
    if (p == '+') r = b + a;
    if (p == '-') r = b - a;
    if (p == '*') r = b * a;
    if (p == '/') r = b / a;
```

```cpp
    num.push(r);  // 结果入栈
}

int main() {
    string s;  // 读入表达式
    cin >> s;

    for (int i = 0; i < s.size(); i++) {
        if(s[i]=='#'||s[i]==' '){
            continue;
        }
        //string不考虑结束与休止,因此跳过
        if (isdigit(s[i])) {  // 数字入栈
            int x = 0, j = i;  // 计算数字
            while (j < s.size() && isdigit(s[j])) {
                x = x * 10 + s[j] - '0';
                j++;
            }
            num.push(x);  // 数字入栈
            i = j - 1;
        }
        else if (s[i] == '(') {  // 左括号入栈
            op.push(s[i]);
        }
        else if (s[i] == ')') {  // 右括号
            while (op.top() != '(')  // 一直计算到左括号
                eval();
            op.pop();  // 左括号出栈
        }
        else {  // 运算符
            while (op.getSize() && h[op.top()] >= h[s[i]])  // 优先级低，先计算
                eval();
            op.push(s[i]);  // 操作符入栈
        }
    }
    while (op.getSize()) eval();  // 剩余的进行计算
    cout << num.top() << endl;  // 输出结果
    return 0;
}
```

总结

- 第一题的链表数据结构很简单,在实验中的题目就写过,但是对于如何获取系统进程的信息,我也是查了很多资料,最后发现需要使用系统自带的API.但是这个API此前没有用过,对于API的调用我使用的也不是很熟悉.在我配置的**gcc**环境中一直报错,但是却没有返回任何信息.查阅各种资料后发现是**gcc**的问题,最后换了vs2022的**msvc**环境才解决.
- 第二题的八皇后上学期就写过,这学期的实验也写过因此很顺利就完成了,结合上学期所学的模板自己尝试实现了**stack**,成就感十足.
- 来到第三题,家谱管理系统的题目要求很多,题目的要求和函数很多,但是数据结构很简单,归根究底还是树的遍历和查找,因为题目很繁琐,所以调用了**vector**,**string**和**map**来存储数据.对于第n代的子孙遍历和之前的实验很像,简单的**bfs**传参数就可以完成了,所以虽然代码很多,但是逻辑很简单.
- 然后是第四题,第四题还是比较有难度的,之前对于平衡二叉排序树只是了解具体的算法,但是对于具体的实现还是没有了解过,自己实现起来以及调用函数的时候也是遇到了很多问题,比如对于四种情况的判断,该如何旋转还是思考了比较久的时间.
- 第五题哈夫曼树的逻辑也是比较简单,统计字符频率,然后建立堆就行,代码量也一般,文件的二进制读写有些忘了,上网查了些资料才写出来.也用到了cpp里的**STL**对于工作量大大的减少.
- 第六题的题目很简单,简单的dfs回溯就完成了
- 第七题的想了很久该怎么读取三个节点,然后转换成图,然后用图实现最短路径和最少乘车次数,最后发现两个都用bfs还是简单一些.
- 第八题是本次课设我觉得最难的一道题,因为之前没有接触过B树,分裂和删除的逻辑很复杂查了很多资料,最后成功复现.
- 第九题的排序之前都写过,对于计算时间的函数上网查了自带的库进行计算.
- 选修题的**prim**和**kruskal**算法以及计算表达式都是很经典的题目,之前都写过,所以很快就写出来了.
- 本次的代码278+135+882+252+190+70+299+400+253+152+168=3079行.这一次的数据结构我写了很久很久,从结课前一周就开始写了,因为学期末课也不是很多,所以一有空就开始写数据结构课设,也查了很多博客和库的文档,学习了不少新知识,例如**STL**,迭代器和模板之前只是听过没有实践过,还有诸如**auto**和智能指针等等新的特性,扩展了自己的眼界.最后,感谢老师和助教的指导,让我能够顺利完成这次课设.