【华为OD机考 统一考试机试C卷】文件缓存系统（C++ Java JavaScript Python C语言）

华为OD机考:统一考试 C卷 + D卷 + B卷 +A卷

2023年11月份，华为官方已经将 华为OD机考：OD统一考试（A卷 / B卷）切换到 OD统一考试（C卷）和 OD统一考试（D卷） 。根据考友反馈：目前抽到的试卷为B卷或C卷/D卷，其中C卷居多 ，按照之前的经验C卷D卷部分考题会复用A卷/B卷题，博主正积极从考过的同学收集C卷和D卷真题，可以查看下面的真题目录。

真题目录：华为OD机考机试 真题目录（C卷 + D卷 + B卷 + A卷） + 考点说明
专栏：2023华为OD机试(B卷+C卷+D卷)（C++JavaJSPy）
华为OD面试真题精选：华为OD面试真题精选
在线OJ：点击立即刷题，模拟真实机考环境 华为OD机考B卷C卷华为OD机考华为OD机考B卷华为OD机试B卷华为OD机试C卷华为OD机考C卷华为OD机考D卷题目华为OD机考C卷/D卷答案华为OD机考C卷/D卷解析华为OD机考C卷和D卷真题华为OD机考C卷和D卷题解

题目描述

请设计一个文件缓存系统，该文件缓存系统可以指定缓存的最大值（单位为字节）。

文件缓存系统有两种操作：

- 存储文件（put）
- 读取文件（get）

操作命令为：

- put fileName fileSize
- get fileName

存储文件是把文件放入文件缓存系统中；

读取文件是从文件缓存系统中访问已存在，如果文件不存在，则不作任何操作。

当缓存空间不足以存放新的文件时，根据规则删除文件，直到剩余空间满足新的文件大小位置，再存放新文件。

具体的删除规则为：文件访问过后，会更新文件的最近访问时间和总的访问次数，当缓存不够时，按照第一优先顺序为访问次数从少到多，第二顺序为时间从老到新的方式来删除文件。

输入描述

第一行为缓存最大值 m（整数，取值范围为 0 < m ≤ 52428800）

第二行为文件操作序列个数 n（0 ≤ n ≤ 300000）

从第三行起为文件操作序列，每个序列单独一行，文件操作定义为：

> op file_name file_size

file_name 是文件名，file_size 是文件大小

输出描述

输出当前文件缓存中的文件名列表，文件名用英文逗号分隔，按字典顺序排序，如：

> a,c

如果文件缓存中没有文件，则输出NONE

备注

1. 如果新文件的文件名和文件缓存中已有的文件名相同，则不会放在缓存中
2. 新的文件第一次存入到文件缓存中时，文件的总访问次数不会变化，文件的最近访问时间会更新到最新时间
3. 每次文件访问后，总访问次数加1，最近访问时间更新到最新时间
4. 任何两个文件的最近访问时间不会重复
5. 文件名不会为空，均为小写字母，最大长度为10
6. 缓存空间不足时，不能存放新文件
7. 每个文件大小都是大于 0 的整数

用例1

输入

```
50
6
put a 10
put b 20
get a
get a
get b
put c 30
```

输出

```
a,c
```

用例2

输入

```
50
1
get file
```

输出

```
NONE
```

解题思路

主要考察“最少使用频率”（Least Frequently Used, LFU）缓存策略的文件缓存系统。这种缓存系统特别适用于需要优先保留最常被访问的项的场景。解题思路和方法如下：

解题思路：

1. **初始化数据结构**：
    - 存储文件及其属性（文件名、大小、访问次数、最后访问时间）。

- 使用（最小堆）维护文件的删除顺序，基于访问次数和最后访问时间。

2. **处理输入**:
   - 接收缓存的最大容量和操作数。
   - 对于每个操作（存储或获取文件），解析并执行相应的逻辑。
3. **缓存操作**:
   - `put` 方法：添加新文件到缓存。如果缓存已满，先移除访问次数最少且最早的文件，然后添加新文件。
   - `get` 方法：从缓存中检索文件，同时更新其访问次数和最后访问时间。
4. **更新和维护缓存**:
   - 每当文件被访问时，更新其在 文件信息 和 最小堆 中的信息。
   - 当缓存空间不足时，根据LFU策略移除文件。

LFU缓存方法：

LFU缓存是一种缓存算法，用于管理有限的资源（如内存）。其核心思想是当需要空间时，优先移除访问频率最低的项。与之相对的是LRU（最近最少使用）缓存，后者基于时间顺序（最近使用的）移除元素。

LFU缓存的关键特点：

- **访问计数**：每个缓存项都有一个与之关联的访问计数器，记录该项被访问的次数。
- **优先级队列**：使用优先级队列（如最小堆）来保持项的顺序，基于访问次数。
- **动态调整**：缓存项的访问计数随着时间的推移动态更新，以反映其使用频率。

C++

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

```

```cpp
using namespace std;

// 用于表示缓存中文件的结构体
struct File {
    int accessCount; // 文件访问次数
    int fileSize;    // 文件大小
    string fileName; // 文件名
    int timeStamp;   // 时间戳

    // 需要定义 > 操作符，因为 priority_queue 在内部用它来进行比较
    bool operator>(const File& other) const {
        if (accessCount == other.accessCount) {
            return timeStamp > other.timeStamp; // 最小堆，用于最近最少使用（LRU）
        }
        return accessCount > other.accessCount; // 最小堆，用于最不频繁使用（LFU）
    }
};

class FileCacheSystem {
private:
    int maxCacheSize; // 缓存最大容量
    int currentCacheSize; // 当前缓存大小
    unordered_map<string, File> cache; // 缓存映射，用于快速查找文件
    priority_queue<File, vector<File>, greater<File>> minHeap; // 最小堆，用于维护文件优先级
    int time; // 时间计数器，用于更新文件时间戳

    // 重建堆 - 这是必要的，因为 priority_queue 不允许直接更新元素
    void rebuildHeap() {
        priority_queue<File, vector<File>, greater<File>> newHeap;
        for (auto& pair : cache) {
            newHeap.push(pair.second);
        }
        minHeap.swap(newHeap); // 用新堆替换旧堆
    }

public:
    FileCacheSystem(int maxSize) : maxCacheSize(maxSize), currentCacheSize(0), time(0) {}

    void put(const string& fileName, int fileSize) {
        if (fileSize > maxCacheSize) return; // 如果文件大小超过缓存容量，则不处理

```

```
50
51          if (cache.find(fileName) != cache.end()) {
52              get(fileName); // 如果文件已存在于缓存中，则更新其访问次数
53              return;
54          }
55
56          // 如果加入新文件后超出缓存容量，需要移除优先级最低的文件
57          while (currentCacheSize + fileSize > maxCacheSize) {
58              File toRemove = minHeap.top();
59              minHeap.pop();
60              cache.erase(toRemove.fileName);
61              currentCacheSize -= toRemove.fileSize;
62          }
63
64          time++; // 更新时间戳
65          File file = {1, fileSize, fileName, time}; // 创建新文件
66          minHeap.push(file); // 将新文件加入最小堆
67          cache[fileName] = file; // 将新文件加入缓存映射
68          currentCacheSize += fileSize; // 更新当前缓存大小
69      }
70
71      void get(const string& fileName) {
72          if (cache.find(fileName) == cache.end()) return; // 如果文件不在缓存中，则不处理
73
74          File& file = cache[fileName];
75          file.accessCount++; // 更新访问次数
76          file.timeStamp = ++time; // 更新时间戳
77          rebuildHeap(); // 更新文件后重建堆
78      }
79
80      vector<string> getCurrentCache() {
81          vector<string> files;
82          for (auto& pair : cache) {
83              files.push_back(pair.first); // 将缓存中的文件名添加到列表
84          }
85          sort(files.begin(), files.end()); // 对文件名进行排序
86          return files;
87      }
88  };
89
90
```

```cpp
int main() {
    int maxCacheSize, operationsCount;
    cin >> maxCacheSize >> operationsCount; // 读取缓存最大容量和操作数

    FileCacheSystem cacheSystem(maxCacheSize); // 创建文件缓存系统实例

    for (int i = 0; i < operationsCount; ++i) {
        string operation, fileName;
        cin >> operation >> fileName; // 读取操作和文件名

        if (operation == "put") {
            int fileSize;
            cin >> fileSize; // 读取文件大小
            cacheSystem.put(fileName, fileSize); // 执行 put 操作
        } else if (operation == "get") {
            cacheSystem.get(fileName); // 执行 get 操作
        }
    }

    vector<string> currentCache = cacheSystem.getCurrentCache(); // 获取当前缓存中的文件列表
    if (currentCache.empty()) {
        cout << "NONE" << endl; // 如果缓存为空，输出 "NONE"
    } else {
        for (size_t i = 0; i < currentCache.size(); ++i) {
            if (i > 0) cout << ","; // 输出文件名之间的逗号
            cout << currentCache[i]; // 输出文件名
        }
        cout << endl; // 输出换行符
    }

    return 0;
}
```

Java

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // 创建扫描器读取输入
```

```
        Scanner scanner = new Scanner(System.in);
        // 读取缓存最大值
        int maxCacheSize = scanner.nextInt();
        // 读取操作数量
        int operationsCount = scanner.nextInt();
        scanner.nextLine(); // 清除行结束符

        // 初始化文件缓存系统
        FileCacheSystem cacheSystem = new FileCacheSystem(maxCacheSize);

        // 循环处理所有操作
        for (int i = 0; i < operationsCount; i++) {
            // 读取操作指令
            String[] input = scanner.nextLine().split(" ");
            String operation = input[0]; // 操作类型
            String fileName = input[1]; // 文件名

            // 如果是存储文件操作
            if ("put".equals(operation)) {
                int fileSize = Integer.parseInt(input[2]); // 文件大小
                cacheSystem.put(fileName, fileSize); // 存储文件
            } else if ("get".equals(operation)) { // 如果是读取文件操作
                cacheSystem.get(fileName); // 读取文件
            }
        }

        // 获取当前缓存中的文件名列表
        List<String> currentCache = cacheSystem.getCurrentCache();
        // 如果列表为空，输出NONE
        if (currentCache.isEmpty()) {
            System.out.println("NONE");
        } else { // 否则，输出文件名列表，用逗号分隔
            System.out.println(String.join(",", currentCache));
        }
    }

    // 文件缓存系统类
    static class FileCacheSystem {
        // 文件类
        private class File {
```

```
        String name; // 文件名
        int size; // 文件大小
        int accessCount; // 访问次数
        long lastAccessTime; // 最近访问时间

        // 文件构造函数
        File(String name, int size, long lastAccessTime) {
            this.name = name;
            this.size = size;
            this.accessCount = 1; // 初始访问次数为1
            this.lastAccessTime = lastAccessTime; // 设置最近访问时间
        }
    }

    // 缓存最大值
    private int maxCacheSize;
    // 当前缓存大小
    private int currentCacheSize = 0;
    // 缓存映射，存储文件名和文件信息
    private HashMap<String, File> cache;
    // 优先队列，用于维护删除顺序
    private PriorityQueue<File> minHeap;
    // 时间戳，用于更新文件的最近访问时间
    private long time;

    // 文件缓存系统构造函数
    public FileCacheSystem(int maxCacheSize) {
        this.maxCacheSize = maxCacheSize; // 设置缓存最大值
        this.cache = new HashMap<>(); // 初始化缓存映射
        // 初始化优先队列，比较器根据访问次数和最近访问时间排序
        this.minHeap = new PriorityQueue<>((a, b) -> {
            if (a.accessCount == b.accessCount) {
                return Long.compare(a.lastAccessTime, b.lastAccessTime);
            }
            return a.accessCount - b.accessCount;
        });
        this.time = 0; // 初始化时间戳
    }

    // 存储文件方法

```

```
    public void put(String fileName, int fileSize) {
        if (fileSize > maxCacheSize) return; // 如果文件大小超过最大缓存，不存储

        if (cache.containsKey(fileName)) {
            get(fileName); // 如果文件已存在，更新访问次数和时间
            return;
        }

        // 当缓存空间不足时，删除访问次数最少且最早访问的文件
        while (currentCacheSize + fileSize > maxCacheSize) {
            File toRemove = minHeap.poll(); // 从优先队列中取出要删除的文件
            if (toRemove != null) {
                cache.remove(toRemove.name); // 从缓存映射中删除
                currentCacheSize -= toRemove.size; // 更新当前缓存大小
            }
        }

        // 创建新文件，添加到缓存映射和优先队列中
        File file = new File(fileName, fileSize, ++time);
        cache.put(fileName, file);
        minHeap.offer(file);
        currentCacheSize += fileSize; // 更新当前缓存大小
    }

    // 读取文件方法
    public void get(String fileName) {
        if (!cache.containsKey(fileName)) return; // 如果文件不存在，不作操作

        // 获取文件信息，更新访问次数和最近访问时间
        File file = cache.get(fileName);
        minHeap.remove(file); // 从优先队列中移除
        file.accessCount++; // 增加访问次数
        file.lastAccessTime = ++time; // 更新最近访问时间
        minHeap.offer(file); // 重新添加到优先队列
    }

    // 获取当前缓存文件名列表方法
    public List<String> getCurrentCache() {
        List<String> files = new ArrayList<>(cache.keySet()); // 获取所有文件名
        Collections.sort(files); // 按字典顺序排序

```

```
            return files;
        }
    }
}
```

javaScript

```javascript
class FileCacheSystem {
    // 构造函数，初始化文件缓存系统
    constructor(maxCacheSize) {
        this.maxCacheSize = maxCacheSize; // 最大缓存大小
        this.currentCacheSize = 0; // 当前缓存大小
        this.cache = {}; // 缓存存储对象，键为文件名，值为文件信息
        this.minHeap = []; // 最小堆数组，用于维护文件的访问优先级
        this.time = 0; // 时间戳，用于记录文件的最后访问时间
    }

    // 比较器函数，用于维护最小堆的顺序
    compare(a, b) {
        if (a.accessCount === b.accessCount) {
            // 如果访问次数相同，则比较最后访问时间
            return a.lastAccessTime - b.lastAccessTime;
        }
        // 否则，比较访问次数
        return a.accessCount - b.accessCount;
    }

    // 最小堆的堆化操作，确保父节点的值小于子节点的值
    minHeapify(index) {
        let smallest = index; // 假设当前索引所在的节点是最小的
        const leftChild = 2 * index + 1; // 左子节点索引
        const rightChild = 2 * index + 2; // 右子节点索引

        // 如果左子节点存在，且小于当前最小节点，则更新最小节点索引
        if (leftChild < this.minHeap.length && this.compare(this.minHeap[leftChild], this.minHeap[smallest]) < 0) {
            smallest = leftChild;
        }
        // 如果右子节点存在，且小于当前最小节点，则更新最小节点索引
        if (rightChild < this.minHeap.length && this.compare(this.minHeap[rightChild], this.minHeap[smallest]) < 0) {
            smallest = rightChild;
```

```
        }

        // 如果最小节点索引有更新，则交换当前节点与最小节点的位置，并递归调用堆化操作
        if (smallest !== index) {
            [this.minHeap[smallest], this.minHeap[index]] = [this.minHeap[index], this.minHeap[smallest]];
            this.minHeapify(smallest);
        }
    }

    // 插入元素到最小堆，维护最小堆的性质
    minHeapInsert(file) {
        this.minHeap.push(file); // 将文件信息插入到最小堆数组的末尾
        let index = this.minHeap.length - 1; // 新插入元素的索引
        let parentIndex = Math.floor((index - 1) / 2); // 新插入元素的父节点索引

        // 当新插入元素的值小于其父节点的值时，交换它们的位置，并更新索引为父节点的索引，继续向上比较
        while (index > 0 && this.compare(this.minHeap[index], this.minHeap[parentIndex]) < 0) {
            [this.minHeap[parentIndex], this.minHeap[index]] = [this.minHeap[index], this.minHeap[parentIndex]];
            index = parentIndex;
            parentIndex = Math.floor((index - 1) / 2);
        }
    }

    // 移除并返回最小堆的顶部元素，即优先级最高（访问次数最少，最早访问）的文件
    minHeapPop() {
        if (this.minHeap.length === 0) return null; // 如果最小堆为空，则返回null
        const minItem = this.minHeap[0]; // 获取最小堆的顶部元素
        this.minHeap[0] = this.minHeap[this.minHeap.length - 1]; // 将最小堆的最后一个元素移动到顶部
        this.minHeap.pop(); // 移除最小堆的最后一个元素
        this.minHeapify(0); // 对新的顶部元素执行堆化操作
        return minItem; // 返回被移除的顶部元素
    }

    // 从最小堆中删除指定元素
    minHeapRemove(file) {
        const index = this.minHeap.findIndex(f => f.name === file.name); // 查找要删除的文件在最小堆中的索引
        if (index === -1) return; // 如果文件不存在于最小堆中，则不执行删除操作

        // 将最小堆的最后一个元素移动到要删除的元素的位置，并执行堆化操作
        this.minHeap[index] = this.minHeap[this.minHeap.length - 1];
```

```
        this.minHeap.pop();
        this.minHeapify(index);
    }

    // 存储文件的方法
    put(fileName, fileSize) {
        if (fileSize > this.maxCacheSize) return; // 如果文件大小超过最大缓存大小，则不执行存储操作

        const currentTime = Date.now(); // 获取当前时间戳
        if (this.cache[fileName]) {
            // 如果文件已经存在于缓存中，则更新文件的访问信息
            this.get(fileName);
            return;
        }

        // 当当前缓存大小加上新文件的大小超过最大缓存大小时，从最小堆中移除优先级最高的文件
        while (this.currentCacheSize + fileSize > this.maxCacheSize) {
            const toRemove = this.minHeapPop();
            if (toRemove) {
                delete this.cache[toRemove.name]; // 从缓存存储对象中删除文件
                this.currentCacheSize -= toRemove.size; // 更新当前缓存大小
            }
        }

        // 创建新文件对象，包含文件名、大小、访问次数和最后访问时间，并将其添加到缓存存储对象和最小堆中
        const file = { name: fileName, size: fileSize, accessCount: 1, lastAccessTime: currentTime };
        this.cache[fileName] = file;
        this.minHeapInsert(file);
        this.currentCacheSize += fileSize; // 更新当前缓存大小
    }

    // 获取文件的方法
    get(fileName) {
        if (!this.cache[fileName]) return; // 如果文件不存在于缓存中，则不执行任何操作

        const file = this.cache[fileName]; // 获取文件信息
        this.minHeapRemove(file); // 从最小堆中删除文件信息
        file.accessCount++; // 更新文件的访问次数
        file.lastAccessTime = Date.now(); // 更新文件的最后访问时间
        this.minHeapInsert(file); // 将更新后的文件信息重新插入到最小堆中

```

```
    }

    // 获取当前缓存中所有文件名的方法
    getCurrentCache() {
        const files = Object.keys(this.cache); // 获取缓存存储对象中所有的键，即文件名
        files.sort(); // 对文件名进行字典序排序
        return files; // 返回排序后的文件名数组
    }
}

// 以下是模拟主函数的部分，用于处理输入和输出
const readline = require('readline'); // 引入readline模块
const rl = readline.createInterface({
    input: process.stdin, // 输入来源为标准输入
    output: process.stdout // 输出目标为标准输出
});

const inputs = []; // 存储输入行的数组
rl.on('line', (line) => {
    inputs.push(line); // 将每行输入添加到数组中
}).on('close', () => {
    // 当输入结束时执行的回调函数
    const maxCacheSize = parseInt(inputs[0]); // 解析最大缓存大小
    const operationsCount = parseInt(inputs[1]); // 解析操作数量
    const cacheSystem = new FileCacheSystem(maxCacheSize); // 创建文件缓存系统实例

    // 循环处理每个操作
    for (let i = 2; i < operationsCount + 2; i++) {
        const input = inputs[i].split(' '); // 分割输入行为操作和参数
        const operation = input[0]; // 操作类型
        const fileName = input[1]; // 文件名

        // 根据操作类型执行相应的方法
        if (operation === 'put') {
            const fileSize = parseInt(input[2]); // 解析文件大小
            cacheSystem.put(fileName, fileSize); // 执行存储文件操作
        } else if (operation === 'get') {
            cacheSystem.get(fileName); // 执行获取文件操作
        }
    }

```

```
157
158     // 获取当前缓存中的所有文件名，并输出
159     const currentCache = cacheSystem.getCurrentCache();
160     if (currentCache.length === 0) {
161         console.log('NONE'); // 如果没有文件，则输出NONE
162     } else {
            console.log(currentCache.join(',')); // 否则输出文件名，用逗号分隔
        }
    });
```

Python

```python
import heapq

class Main:
    def __init__(self):
        # 创建输入扫描器
        self.cacheSystem = None

    def main(self):
        # 读取缓存最大值
        maxCacheSize = int(input())
        # 读取操作数量
        operationsCount = int(input())

        # 初始化文件缓存系统
        self.cacheSystem = FileCacheSystem(maxCacheSize)

        # 循环处理所有操作
        for _ in range(operationsCount):
            # 读取操作指令
            input_line = input().split(" ")
            operation = input_line[0]  # 操作类型
            fileName = input_line[1]  # 文件名

            # 如果是存储文件操作
            if operation == "put":
                fileSize = int(input_line[2])  # 文件大小
                self.cacheSystem.put(fileName, fileSize)  # 存储文件
            elif operation == "get":  # 如果是读取文件操作
```

```
                self.cacheSystem.get(fileName)  # 读取文件

        # 获取当前缓存中的文件名列表
        currentCache = self.cacheSystem.getCurrentCache()
        # 如果列表为空，输出NONE
        if not currentCache:
            print("NONE")
        else:  # 否则，输出文件名列表，用逗号分隔
            print(",".join(currentCache))


class FileCacheSystem:
    def __init__(self, maxCacheSize):
        # 缓存最大值
        self.maxCacheSize = maxCacheSize
        # 当前缓存大小
        self.currentCacheSize = 0
        # 缓存映射，存储文件名和文件信息
        self.cache = {}
        # 优先队列，用于维护删除顺序
        self.minHeap = []
        # 时间戳，用于更新文件的最近访问时间
        self.time = 0

    def put(self, fileName, fileSize):
        # 存储文件方法
        if fileSize > self.maxCacheSize:
            return  # 如果文件大小超过最大缓存，不存储

        if fileName in self.cache:
            self.get(fileName)  # 如果文件已存在，更新访问次数和时间
            return

        # 当缓存空间不足时，删除访问次数最少且最早访问的文件
        while self.currentCacheSize + fileSize > self.maxCacheSize:
            toRemove = heapq.heappop(self.minHeap)  # 从优先队列中取出要删除的文件
            del self.cache[toRemove[2]]  # 从缓存映射中删除
            self.currentCacheSize -= toRemove[1]  # 更新当前缓存大小

        # 创建新文件，添加到缓存映射和优先队列中
```

```
        self.time += 1
        file = (1, fileSize, fileName, self.time)  # 访问次数，文件大小，文件名，最近访问时间
        self.cache[fileName] = file
        heapq.heappush(self.minHeap, file)
        self.currentCacheSize += fileSize  # 更新当前缓存大小

    def get(self, fileName):
        # 读取文件方法
        if fileName not in self.cache:
            return  # 如果文件不存在，不作操作

        # 获取文件信息，更新访问次数和最近访问时间
        file = self.cache[fileName]
        self.minHeap.remove(file)  # 从优先队列中移除
        self.time += 1
        new_file = (file[0] + 1, file[1], file[2], self.time)  # 更新文件信息
        heapq.heappush(self.minHeap, new_file)  # 重新添加到优先队列
        self.cache[fileName] = new_file  # 更新缓存映射

    def getCurrentCache(self):
        # 获取当前缓存文件名列表方法
        return sorted(self.cache.keys())  # 获取所有文件名并按字典顺序排序


if __name__ == "__main__":
    Main().main()
```

C语言

```
// 引入标准输入输出库、标准库、字符串操作库和时间库
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define MAX_FILES 1000 // 定义最大文件数量常量

// 定义文件结构体，包含文件名、大小、访问次数和最后访问时间
typedef struct {
    char name[256]; // 文件名，假设最大长度为255
```

```
    int size;       // 文件大小，单位为字节
    int accessCount; // 访问次数，用于记录文件被访问的频率
    long lastAccessTime; // 最后访问时间，用于记录文件最后被访问的时间戳
} File;

// 定义文件缓存系统结构体，包含最大缓存大小、当前缓存大小、缓存数组和最小堆数组
typedef struct {
    int maxCacheSize; // 最大缓存大小，单位为字节
    int currentCacheSize; // 当前缓存大小，单位为字节
    File *cache[MAX_FILES]; // 缓存数组，用于存储文件指针
    File *minHeap[MAX_FILES]; // 最小堆数组，用于维护文件的优先级队列
    int minHeapSize; // 最小堆的当前大小，用于记录最小堆中元素的数量
} FileCacheSystem;

// 比较器函数，用于比较两个文件的访问优先级
int compare(File *a, File *b) {
    if (a->accessCount == b->accessCount) {
        // 如果访问次数相同，则比较最后访问时间，时间早的优先级高
        return (int)(a->lastAccessTime - b->lastAccessTime);
    }
    // 如果访问次数不同，则访问次数少的优先级高
    return a->accessCount - b->accessCount;
}

// 最小堆的堆化操作，确保父节点的值小于子节点的值
void minHeapify(FileCacheSystem *fcs, int index) {
    int smallest = index; // 当前节点索引
    int leftChild = 2 * index + 1; // 左子节点索引
    int rightChild = 2 * index + 2; // 右子节点索引

    // 如果左子节点存在且小于当前节点，更新最小值索引
    if (leftChild < fcs->minHeapSize && compare(fcs->minHeap[leftChild], fcs->minHeap[smallest]) < 0) {
        smallest = leftChild;
    }
    // 如果右子节点存在且小于当前节点，更新最小值索引
    if (rightChild < fcs->minHeapSize && compare(fcs->minHeap[rightChild], fcs->minHeap[smallest]) < 0) {
        smallest = rightChild;
    }

    // 如果最小值索引不是当前节点，交换并递归调用堆化操作
```

```
    if (smallest != index) {
        File *temp = fcs->minHeap[smallest];
        fcs->minHeap[smallest] = fcs->minHeap[index];
        fcs->minHeap[index] = temp;
        minHeapify(fcs, smallest);
    }
}

// 插入元素到最小堆，维护最小堆的性质
void minHeapInsert(FileCacheSystem *fcs, File *file) {
    // 将新文件插入到最小堆末尾
    fcs->minHeap[fcs->minHeapSize] = file;
    int index = fcs->minHeapSize; // 新插入元素的索引
    fcs->minHeapSize++; // 堆大小增加

    // 从当前节点开始，向上调整堆
    int parentIndex = (index - 1) / 2; // 父节点索引
    while (index > 0 && compare(fcs->minHeap[index], fcs->minHeap[parentIndex]) < 0) {
        // 如果当前节点小于父节点，则交换它们的位置
        File *temp = fcs->minHeap[parentIndex];
        fcs->minHeap[parentIndex] = fcs->minHeap[index];
        fcs->minHeap[index] = temp;
        // 更新当前节点和父节点的索引，继续向上调整
        index = parentIndex;
        parentIndex = (index - 1) / 2;
    }
}

// 移除并返回最小堆的顶部元素
File *minHeapPop(FileCacheSystem *fcs) {
    if (fcs->minHeapSize == 0) return NULL; // 如果堆为空，则返回NULL
    File *minItem = fcs->minHeap[0]; // 获取堆顶元素
    // 将堆的最后一个元素移动到堆顶
    fcs->minHeap[0] = fcs->minHeap[fcs->minHeapSize - 1];
    fcs->minHeapSize--; // 堆大小减少
    minHeapify(fcs, 0); // 对新的堆顶元素进行堆化操作
    return minItem; // 返回原堆顶元素
}

// 从最小堆中删除指定元素

```

```
void minHeapRemove(FileCacheSystem *fcs, File *file) {
    int index = -1;
    // 遍历最小堆，找到要删除的元素的索引
    for (int i = 0; i < fcs->minHeapSize; i++) {
        if (strcmp(fcs->minHeap[i]->name, file->name) == 0) {
            index = i;
            break;
        }
    }
    if (index == -1) return; // 如果未找到，直接返回

    // 将堆的最后一个元素移动到要删除的元素的位置
    fcs->minHeap[index] = fcs->minHeap[fcs->minHeapSize - 1];
    fcs->minHeapSize--; // 堆大小减少
    minHeapify(fcs, index); // 对新的元素进行堆化操作
}

// 存储文件的方法
void put(FileCacheSystem *fcs, char *fileName, int fileSize) {
    if (fileSize > fcs->maxCacheSize) return; // 如果文件大小超过最大缓存大小，直接返回

    long currentTime = time(NULL); // 获取当前时间
    // 遍历缓存数组，检查文件是否已存在
    for (int i = 0; i < MAX_FILES; i++) {
        if (fcs->cache[i] != NULL && strcmp(fcs->cache[i]->name, fileName) == 0) {
            // 如果文件已存在，则更新文件的访问信息
            get(fcs, fileName);
            return;
        }
    }

    // 如果当前缓存加上新文件大小超过最大缓存大小，则移除优先级最低的文件
    while (fcs->currentCacheSize + fileSize > fcs->maxCacheSize) {
        File *toRemove = minHeapPop(fcs);
        if (toRemove != NULL) {
            // 遍历缓存数组，找到并释放要移除的文件
            for (int i = 0; i < MAX_FILES; i++) {
                if (fcs->cache[i] == toRemove) {
                    fcs->currentCacheSize -= toRemove->size; // 更新当前缓存大小
                    free(fcs->cache[i]); // 释放文件内存

```

```
                    fcs->cache[i] = NULL; // 将缓存位置置为空
                    break;
                }
            }
        }
    }

    // 为新文件分配内存并初始化
    File *file = (File *)malloc(sizeof(File));
    strcpy(file->name, fileName); // 复制文件名
    file->size = fileSize; // 设置文件大小
    file->accessCount = 1; // 初始化访问次数为1
    file->lastAccessTime = currentTime; // 设置最后访问时间为当前时间

    // 将新文件添加到缓存数组中的空位置
    for (int i = 0; i < MAX_FILES; i++) {
        if (fcs->cache[i] == NULL) {
            fcs->cache[i] = file;
            break;
        }
    }

    // 将新文件插入到最小堆中
    minHeapInsert(fcs, file);
    fcs->currentCacheSize += fileSize; // 更新当前缓存大小
}

// 获取文件的方法
void get(FileCacheSystem *fcs, char *fileName) {
    // 遍历缓存数组，查找文件
    for (int i = 0; i < MAX_FILES; i++) {
        if (fcs->cache[i] != NULL && strcmp(fcs->cache[i]->name, fileName) == 0) {
            File *file = fcs->cache[i];
            // 从最小堆中移除文件
            minHeapRemove(fcs, file);
            // 更新文件的访问次数和最后访问时间
            file->accessCount++;
            file->lastAccessTime = time(NULL);
            // 将更新后的文件重新插入到最小堆中
            minHeapInsert(fcs, file);

```

```
            break;
        }
    }
}

// 初始化文件缓存系统
void initFileCacheSystem(FileCacheSystem *fcs, int maxCacheSize) {
    fcs->maxCacheSize = maxCacheSize; // 设置最大缓存大小
    fcs->currentCacheSize = 0; // 初始化当前缓存大小为0
    fcs->minHeapSize = 0; // 初始化最小堆大小为0
    // 清空缓存数组和最小堆数组
    memset(fcs->cache, 0, sizeof(fcs->cache));
    memset(fcs->minHeap, 0, sizeof(fcs->minHeap));
}

// 主函数，用于处理输入和输出
int main() {
    int maxCacheSize, operationsCount;
    // 读取最大缓存大小和操作次数
    scanf("%d %d", &maxCacheSize, &operationsCount);

    FileCacheSystem fcs;
    // 初始化文件缓存系统
    initFileCacheSystem(&fcs, maxCacheSize);

    char operation[4], fileName[256];
    int fileSize;
    // 根据操作次数，循环读取操作和文件信息
    for (int i = 0; i < operationsCount; i++) {
        scanf("%s %s", operation, fileName);
        if (strcmp(operation, "put") == 0) {
            // 如果是put操作，读取文件大小并存储文件
            scanf("%d", &fileSize);
            put(&fcs, fileName, fileSize);
        } else if (strcmp(operation, "get") == 0) {
            // 如果是get操作，获取文件
            get(&fcs, fileName);
        }
    }


```

```
    // 输出当前缓存中的所有文件名
    int isEmpty = 1; // 标记缓存是否为空
    for (int i = 0; i < MAX_FILES; i++) {
        if (fcs.cache[i] != NULL) {
            if (isEmpty) {
                isEmpty = 0; // 如果找到第一个文件，更新标记
                printf("%s", fcs.cache[i]->name); // 输出第一个文件名
            } else {
                printf(",%s", fcs.cache[i]->name); // 输出后续文件名，用逗号分隔
            }
        }
    }
    if (isEmpty) {
        printf("NONE\n"); // 如果缓存为空，输出NONE
    } else {
        printf("\n"); // 输出换行符
    }

    // 释放分配的内存
    for (int i = 0; i < MAX_FILES; i++) {
        if (fcs.cache[i] != NULL) {
            free(fcs.cache[i]); // 释放文件内存
        }
    }

    return 0; // 程序结束
}
```

文章目录

LFU缓存方法：

C++

Java

javaScript

Python

C语言