# 2020 CCF 大学生计算机系统与程序设计竞赛

## CCF CCSP 2020

**时间：**2020 年 10 月 17 日 09:00 ~ 15:00

| 题目名称 | 办签证 | 给他文件系统 | 垃圾回收器 |
| --- | --- | --- | --- |
| 题目类型 | 传统型 | 传统型 | 传统型 |
| 输入 | 标准输入 | 标准输入 | 标准输入 |
| 输出 | 标准输出 | 标准输出 | 标准输出 |
| 每个测试点时限 | 1.0 秒 | 3.0 秒 | 30.0 秒 |
| 内存限制 | 512 MiB | 2 GiB | 2 GiB |
| 子任务数目 | 10 | 12 | 20 |
| 测试点是否等分 | 否 | 否 | 否 |

# 办签证（Visa）

## 【题目描述】

世界那么大，我想去看看。

——顿顿

顿顿想欣赏国外美丽风光，所以他计划前往某国领事馆办理签证。请你为顿顿规划符合要求的出行路线。

如图 1 所示，顿顿行动范围可以表示为一张有向图 $G = (V, E)$。其中，$V$ 为城市的集合，$E$ 为城市间民航航班的集合。两个城市间往返方向各最多拥有一个航班，顿顿可以乘飞机从一个城市前往另一个城市。

$V$ 中包含三种城市：顿顿出发的城市 $v_0$、领事馆所在的城市 $U = \{v_1, \ldots, v_{|U|}\} \subset V$ 和中转城市 $V - U - \{v_0\}$。不同城市领事馆办理签证开销各不相同。对于领事馆所在城市 $v \in U$，在该城市办理签证所需开销记为 $c_{\text{visa}}(v)$。

不同航班机票价格各不相同。对于出发城市为 $v_i$ 到达城市为 $v_j$ 的航班 $e_{i,j} = (v_i, v_j) \in E$，其机票价格记为 $c_{\text{air}}(e_{i,j})$。同时，由于航空管制，各航班有不同的延误概率，记为 $P_{\text{delay}}(e_{i,j}) \in [0, 1]$。

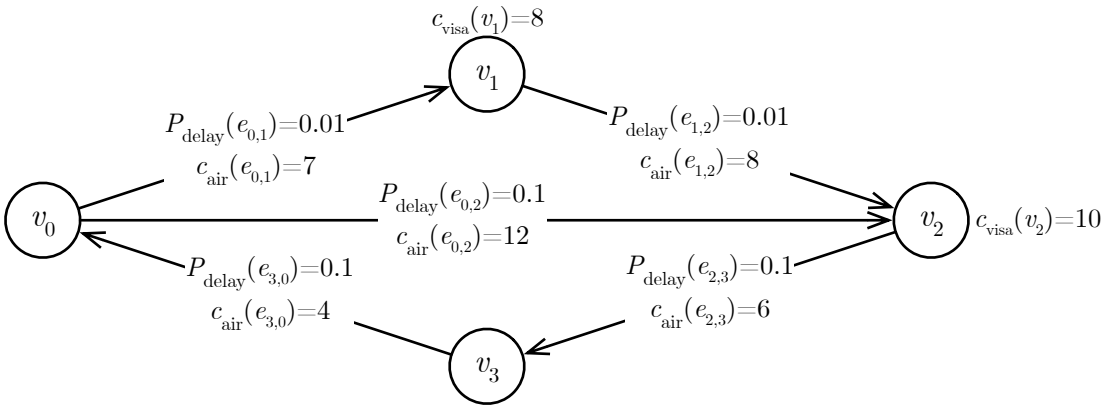


图 1: 顿顿的行动范围 $G$ 示意图

顿顿想在有限预算 $C$ 内从其所在城市 $v_0$ 出发前往任意一个领事馆所在城市 $v_x \in U$，完成签证办理并返回 $v_0$。请你为他选择一个办理签证的城市以及一条开销不超过预算、遭遇航班延误概率最小的出行路线。即

$$\min_{p=(v_{i_1},\ldots,v_{i_n})} 1 - \prod_{\forall 1 \le k < n} (1 - P_{\text{delay}}(e_{i_k,i_{k+1}})),$$

其中

$$v_{i_1} = v_{i_n} = v_0,$$

$$\forall 1 \le k \le n : v_{i_k} \in V,$$

$$\exists 1 < k < n : v_{i_k} = v_x \in U,$$

$$\forall 1 \le k < n : e_{i_k,i_{k+1}} \in E,$$

$$c_{\text{visa}}(v_x) + \sum_{1 \le k < n} c_{\text{air}}(e_{i_k,i_{k+1}}) \le C.$$

## 【输入格式】

从标准输入读入数据。

第一行输入包含四个用空格分隔的正整数，依次为城市个数 $|V|$，包含领事馆的城市个数 $|U|$，航班个数 $|E|$，以及顿顿的预算 $C$。

随后 $|U|$ 行输入依次对应领事馆所在城市 $v_1, \ldots, v_{|U|}$，每行包含一个正整数，为办理签证所需开销 $c_{\text{visa}}(v_1), \ldots, c_{\text{visa}}(v_{|U|})$。

随后 $|E|$ 行输入每行代表一个航班，每行包含四个用空格分隔的数字，依次为：

1. 出发城市编号，为自然数，范围为 $0, \ldots, |V| - 1$，分别对应城市 $v_0, \ldots, v_{|V|-1}$；
2. 到达城市编号，为自然数，范围为 $0, \ldots, |V| - 1$，分别对应城市 $v_0, \ldots, v_{|V|-1}$；
3. 航班延误概率，为浮点数，范围为 $[0, 1]$；
4. 航班的机票价格，为正整数。

## 【输出格式】

输出到标准输出。

第一行输出顿顿办签证城市的编号，即 $x$。

第二行输出顿顿的总开销，即 $c_{\text{visa}}(v_x) + \sum_{1 \le k < n} c_{\text{air}}(e_{i_k,i_{k+1}})$。

第三行输出顿顿的延误概率，即 $1 - \prod_{1 \le k < n}(1 - P_{\text{delay}}(e_{i_k,i_{k+1}}))$。

第四行输出顿顿的出行路线，用一个用空格分隔的城市编号序列表示，即 $i_1\ i_2\ \ldots\ i_n$。注意，$x$ 应包含于该序列中，并且该序列头尾编号应为 0。

## 【样例 1 输入】

```
4 2 5 33
8
10
0 1 0.01 7
0 2 0.1 12
1 2 0.01 8
2 3 0.1 6
3 0 0.1 4
```

【样例 1 输出】

```
1
33
0.206119
0 1 2 3 0
```

【样例 1 解释】

样例输入如图 1 所示。图中从 $v_0$ 出发，共有三种可行出行路线：

- 选择在 $v_1$ 办签证，路线为 $(v_0, v_1, v_2, v_3, v_0)$，开销为 33，遭遇航班延误概率为 0.206119；
- 选择在 $v_2$ 办签证，路线为 $(v_0, v_1, v_2, v_3, v_0)$，开销为 35，遭遇航班延误概率为 0.206119；
- 选择在 $v_2$ 办签证，路线为 $(v_0, v_2, v_3, v_0)$，开销为 32，遭遇航班延误概率为 0.271。

由于预算为 33，从而最优方案为选择 $v_1$，出行路线为 $(v_0, v_1, v_2, v_3, v_0)$。

【样例 2】

见题目目录下的 ***2.in*** 与 ***2.ans***。这是一个稍大的测试用例。

【子任务】

本题目中，测试共包含 4 个测试集，共 10 个测试点，其中每个测试点分值为 10 分，每个测试集的**规模**、**分值**如下表所示：

| 测试集 | 规模 | 总分值 |
|---|---|---|
| 1 | $\|V\| \leq 2 * 10^3, \|E\| \leq 2 * 10^5$ | 30 |
| 2 | $\|V\| \leq 2 * 10^2, C \leq 2 * 10^2, \|E\| \cdot C \leq 1 * 10^5$ | 20 |
| 3 | $\|V\| \leq 1 * 10^3, C \leq 1 * 10^3, \|E\| \cdot C \leq 1 * 10^7$ | 20 |
| 4 | $\|V\| \leq 2 * 10^3, C \leq 2 * 10^3, \|E\| \cdot C \leq 1 * 10^8$ | 30 |

其中，在**测试集 1** 中任意从 $v_0$ 出发前往某地办理签证并返回 $v_0$ 的概率最佳路线累计开销均小于或等于 $C$，意味着**预算限制不产生实际作用**；

此外，所有子任务测试数据**均为无重边的强连通图**，并且**保证至少存在一条满足预算限制的路径**。

## 【提示】

- 出行路线的延误概率为浮点数，故我们接受 $\pm 10^{-4}$ 的误差。
- 当测试点存在多种最优出行路线时，输出任意一条即可。
- 后三个测试集的算法也适用于测试集 1。

# 给他文件系统（GeetFS）

## 【题目背景】

自从被安利了版本控制软件 Git，顿顿沉迷其中无法自拔。恰好最近学习了文件系统的一些知识，于是顿顿打算在办签证的途中实现一个类似于 Git 的特殊“文件系统”GeetFS。

## 【题目描述】

### 基本布局

普通的文件系统使用存储设备保存用户文件，
但是 *GeetFS* 不是一个普通的文件系统。
——顿顿

GeetFS 是一个基于内存的文件系统，其使用内存中的结构来保存和管理用户的文件数据。图 2 展示了 GeetFS 的工作流程和 GeetFS 保存数据的基本方式。在使用 GeetFS 时，用户发送请求给 GeetFS。GeetFS 会解析用户的请求，在内存数据结构中进行操作，完成用户请求。

**GeetFS 中的一些基本概念**

- 文件：与普通文件系统中的文件概念相同，GeetFS 中的文件可以保存数据。GeetFS 中的文件支持读取、写入、删除。写入一个不存在（或此前已被删除）的文件会自动创建该文件；
- 文件名：每个文件拥有一个文件名，为一个长度不超过 128 的字符串。文件名的内容仅包含大小写英文字母（`A~Z` 和 `a~z`）和数字（`0~9`）。顿顿认为 GeetFS 的用户都非常善良，所以他们能够保证每个 GeetFS 命令中的文件名均满足上述规定，GeetFS 在实现的时候无需进行检查。如图 2 中的 `file1` 表示一个名为 `file1` 的文件。
- 暗文件：为了表示文件的删除，GeetFS 中引入了暗文件的概念。对于一个名为 `filename` 的文件，其暗文件名称为 `-filename`（即在文件名前增加了一个负号 `-`）。由于在用户创建的文件名中不允许出现 `-`，暗文件的文件名与普通文件的文件名并不会混淆。暗文件的文件大小为 0，其中不可保存数据。其仅表示名为 `filename` 的文件被删除。一个文件与其暗文件不应同时出现在同一个提交中，也不应同时出现在暂存区中（关于提交和暂存区的概念请看下面两条）。如图 2 中的 `-file2` 为一个暗文件，表示对文件 `file2` 的删除。

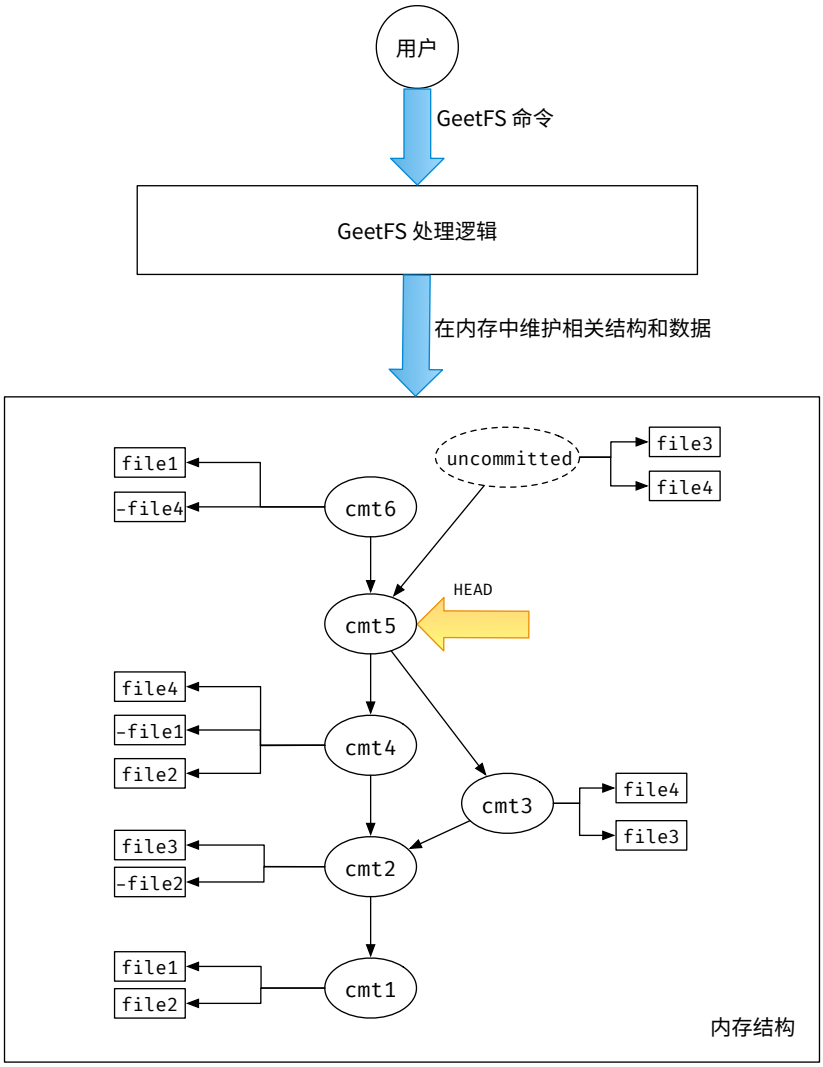


图 2: GeetFS 的工作流程与数据保存方法

- 暂存区：用户所有的修改，包括文件写入、删除等，在未提交时，均保存在暂存区（即图 2 中的 `uncommitted` 结构，包括虚线椭圆及其右侧的文件）。其中文件的写入（包括创建）以文件的形式保存，而文件的删除以暗文件的形式保存。
- 提交：与在 Git 中类似，一个提交表示 GeetFS 的一个历史状态。一般来说，提交由 `commit` 命令创建，GeetFS 将当前暂存区中的所有修改保存下来，并赋予一个唯一的提交名，成为一个提交。提交名的命名要求与文件名相同，且无需进行格式检查。除了 `commit` 命令外，用户还可以通过 `merge` 命令创建一个提交。通过 `merge` 命令创建的提交将两个现有提交进行合并。提交的创建在后文中有具体描述。除了 GeetFS 中的第一个提交不存在父提交之外，其余每个提交拥有一个父提交（由 `commit` 命令创建）或者两个父提交（由 `merge` 命令创建）。如图 2 中的 `cmt1` 表示一个名为 `cmt1` 的提交。
- `HEAD`：与 Git 中类似，`HEAD` 表示当前的头部，指向头部提交。用户当前能够访问哪些文件，以及文件的内容，取决于当前缓存区中的内容以及当前头部所指向

的提交。

- GeetFS 命令：用户使用 GeetFS 命令对 GeetFS 的内容进行操作。命令只能通过标准输入传递给 GeetFS，除了 `write` 命令占据两行之外，其他 GeetFS 均只占一行（即以换行符结尾）。

**GeetFS 的存储结构** GeetFS 中保存了一个头部，一个暂存区结构，和若干个提交结构。

- 头部（`HEAD`）：GeetFS 中有且仅有一个头部，为指向当前头部提交的一个指针或者引用，当 GeetFS 中不存在任何提交时，头部为空。
- 暂存区结构（`uncommitted`）：GeetFS 中有且仅有一个暂存区结构。暂存区结构中包含了所有还未提交的修改。在此结构中，应该保存所有被修改（包括创建）文件的名称、大小和内容。对于删除的文件，该结构中应当保存对应的暗文件的信息。注意暗文件的大小为 0，不保存数据，因此只有暗文件的文件名是有意义的信息。
- 提交结构：GeetFS 中可以有零个或者多个提交结构，保存在元数据结构之中。暂存区结构中的内容在提交后，变为提交结构。一旦生成，提交结构中的内容是不可修改的。

**初始状态** 一个刚刚被创建出来的 GeetFS 文件系统只包括一个空头部和一个空的暂存区结构（即其中没有任何文件或暗文件）。

### 基本操作

普通的文件系统使用目录组织文件，
但是 *GeetFS* 不是一个普通的文件系统。
——顿顿

作为一个“与 (zhu) 众 (yao) 不 (shi) 同 (lan)”的文件系统，GeetFS 只支持对文件的操作，并不支持目录（文件夹），同时 GeekFS 支持的文件操作只有三个：写入、读取和删除。这三个操作均依赖于文件的查找。

**文件的查找** 查找一个文件 `X` 的流程如图所示，具体规则如下：1. 如果暂存区中能够查找到 `X` 文件，则找到了该文件；2. 如果暂存区中不存在 `X` 文件，但是存在其暗文件 `-X`，则表示目标文件被删除，返回文件已被删除；3. 如果在暂存区中，既不存在 `X` 文件，也不存在其暗文件，则在暂存区中无法确定是否存在该文件，需继续查找头部所指向的提交。如果头部为空，则文件不存在；4. 在一个提交中查找文件的方法与在暂存区中查找的方法相同，即先后检查目标文件和其暗文件。如果该提交中依然无法确定是否存在该文件，则继续查找该提交的父提交，直至能够确定目标文件的存在性，或父提交不存在，则该文件不存在。对于具有两个父提交的提交，其查找方法在后文 `merge` 命令中给出。

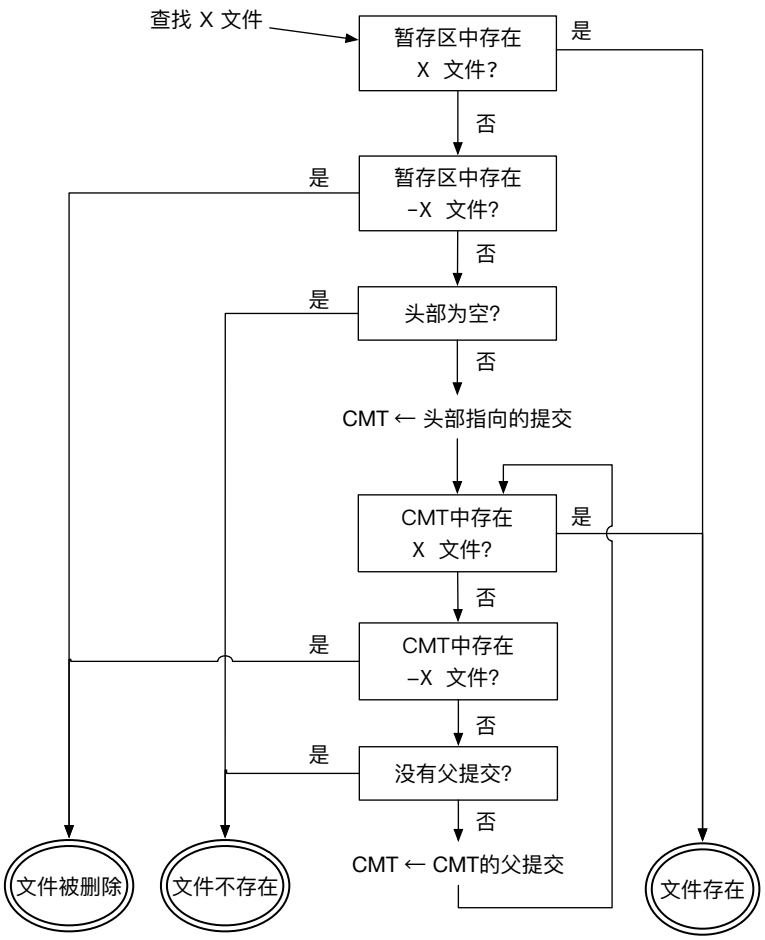


图 3: GeetFS 中查找文件的流程

**写入命令** 共包括两行输入，其中第一行格式为 `write <filename> <offset> <len>`（其中尖括号表示参数，下同），表示向文件 `<filename>` 中写入数据，写入的数据长度为 `<len>` 个字节，写入的开始位置为文件的 `<offset>` 位置（注意，在文件的位置 0 写入 1 个字符，写入的是文件的第 1 个字符）。如果 `<offset>` 的位置超出了该文件的大小，则从文件当前末尾到 `<offset>` 之间的区域使用 ASCII 字符点（`.`）进行填充。

在此行命令之后的一行，用户会输入 `<len>` 个字节作为文件内容，（注意结尾会有一个换行符 \n，不算做文件内容）。用户输入的内容中不包含换行字符，但是**可能包含空格**。

在执行写入命令时，GeetFS 首先按照前述方法查找该文件，并根据不同情况进行不同操作：

- 如果在暂存区中该文件被找到，则直接进行写入操作；
- 如果在某个提交中找到了该文件，则先将找到的文件拷贝到暂存区中（即在暂存区中创建该文件，并将找到的文件的内容拷贝到新文件中），再在暂存区中的文件中进行写入操作；
- 如果查找结果为文件被删除，如果是在暂存区中被删除，则删除暂存区中对应的暗文件，并在暂存区中创建该文件后进行写入操作；

- 如果是在某个提交中被删除，则在暂存区中创建该文件后进行写入操作；
- 如果该文件不存在，则在暂存区中创建该文件，并进行写入操作。

写入命令不产生输出。

**读取命令** 共占一行，格式为 `read <filename> <offset> <len>`，表示从文件 `<filename>` 的 `<offset>` 位置开始读取此后的 `<len>` 个字节。对于超出文件当前大小的部分，每个超出的字节以一个 ASCII 字符点（`.`）替代。在执行时，GeetFS 首先按前述方法查找该文件，如果能找到文件，则输出文件中对应的内容；如果文件不存在或者文件被删除，则输出 `<len>` 个 ASCII 字符点（`.`）。文件读取命令的输出共占一行，因此在文件内容后应有换行（\n）字符，格式也可以参考样例。

**删除命令** 占一行，格式为 `unlink <filename>`，表示删除名为 `<filename>` 的文件。如果 GeetFS 中无法找到该文件或该文件被删除，则什么都不做。如果能够找到该文件，则在暂存区中添加该文件的暗文件。如果目标文件是在暂存区中被找到的，则还需要从暂存区中删除目标文件，只保留其暗文件。

删除命令不产生输出。

注意，删除操作并不能抵消文件创建操作的效果。在文件 `X` 不存在的情况下，用户可以首先创建文件 `X`，之后将文件 `X` 删除。在删除后，暂存区中会存有一个 `X` 的暗文件（`-X`），与文件 `X` 被创建前的状态不同。

**列举命令** 占一行，格式为 `ls`，输出在当前的暂存区和当前的头部状态下，用户能够读到的文件（即可以查找到的文件，不包括暗文件）个数，以及其中按字典序排列，名字最小的文件名和名字最大的文件名。文件个数和两个文件名之间以一个空格隔开，共占一行。如果用户能读到的文件数为 0，则只需输出数字 0，占一行，无需给出文件名。列举命令的输出共占一行，因此在列举内容之后应有换行（\n）字符，具体可以参考样例。

### 高级操作

普通的文件系统仅仅是一个文件系统，
但是 *GeetFS* 不是一个普通的文件系统。
——顿顿

作为一个有 Git 情节的文件系统，GeetFS 当然不仅仅支持标准的文件系统接口。其还支持一系列与 Git 操作相似的高级命令。

**提交命令（`commit`）** 占一行，格式为 `commit <cmtname>`。

`commit` 命令将暂存区中的修改进行提交，其接受一个字符串类型参数 `<cmtname>`，为新提交的名称。在进行提交时，GeetFS 将当前的暂存区 `uncommitted` 重命名为给定的提交名称，并更新元数据信息。新的提交（`<cmtname>`）的父提交为此时头部所指向

的提交。如果此时头部为空，则新提交没有父提交。此后，GeetFS 将更新头部，让其指向刚刚创建的新提交（<cmtname>）。最后，GeetFS 还会创建一个新的空暂存区，用于保存此后的修改。

注意，如果在提交时暂存区为空，或名为 <cmtname> 的提交已经存在，则该命令执行失败，GeetFS 中不产生任何修改。提交命令不产生任何输出。

**切换命令（checkout）** 占一行，格式为 checkout <cmtname>。

checkout 接受一个参数，为提交名 <cmtname>。该命令将当前的头部指向 <cmtname>。在支持该命令之后，提交之间的关系可能会“分叉”。checkout 命令不一定会成功。若在进行 checkout 时，暂存区不为空，或者名为 <cmtname> 的提交不存在，则 checkout 命令执行失败，GeetFS 中不应产生任何修改。

**合并命令（merge）** 占一行，格式为 merge <mergee> <cmtname>。

merge 命令接受两个参数，分别为需要合并的提交名 <mergee> 和新提交的名字 <cmtname>。假设此时头部指向的提交为 headcmt，该命令将 <mergee> 中的内容合并到提交 headcmt 之上。具体来说，GeetFS 会创建一个新的提交，名为 <cmtname>，其两个父提交为 headcmt 和 mergee。由 merge 命令创建的提交中不包含任何文件和数据，只记录了两个父提交，表示这两个父提交的内容在逻辑上进行了合并。

注意，如果在进行 merge 时满足以下任何一个条件，则 merge 执行失败，GeetFS 中不产生任何修改。

- 失败条件 1：暂存区不为空；
- 失败条件 2：<mergee> 与 headcmt 为同一个提交；
- 失败条件 3：名为 <mergee> 的提交不存在。

支持 merge 命令会影响文件查找的规则：在支持 merge 命令后，一个提交（cmt）可以有两个不同的父提交（cmt.parent1 和 cmt.parent2）。在进行文件查找时，若在 cmt 中无法确定目标文件是否存在，则 GeetFS 需要通过 cmt.parent1 和 cmt.parent2 两个父提交分别进行文件查找。

- 若通过两个父提交均无法找到目标文件或其暗文件，则表示要查找的文件不存在；
- 若仅能通过其中一个父提交找到该目标文件或其暗文件，则以此找到的文件作为文件查找的结果；
- 若通过两个父提交均能找到该目标文件或其暗文件，则根据所找到的两个文件的所在提交的创建时间进行选择，取创建时间最近（最大）的文件作为文件查找的结果。如果所找到的两个文件为同一个文件（即在同一个提交中），则以此文件作为文件查找的结果。

### 【输入格式】

从标准输入读入数据。

输入的第一行为一个数字 N，表示该测试中的命令总个数。从第二行开始，标准输入共包含 N 个命令。注意，每个写入命令 `write` 共占两行，其中第一行为写入命令以及其参数，第二行为需要写入的文件内容。文件内容中不会包含换行字符，但是可能会包含空格。其他每个命令占一行，具体格式在前文已经给出。命令均符合相应格式，文件名和提交名均符合规范，读写命令中的长度均大于 0，因此无需对命令格式进行错误处理。

### 【输出格式】

输出到标准输出。

根据每个命令的要求进行相应输出。

### 【样例 1 输入】

```
10
write file1 5 2
78
write file2 7 4
abcd
read file1 0 10
ls
read file2 4 10
unlink file2
ls
read file2 3 4
write file2 1 2
12
read file2 0 4
```

### 【样例 1 输出】

```
.....78...
2 file1 file2
...abcd...
```

```
1 file1 file1
....
.12.
```

【样例 1 解释】

此样例为简单的文件读写测试。

【样例 2 输入】

```
22
write file1 3 2
ab
commit cmt1
write file2 2 4
cdef
read file1 0 10
ls
unlink file1
commit cmt2
ls
checkout cmt1
read file1 0 10
write file1 6 2
gh
write file3 2 3
ijk
commit cmt3
ls
checkout cmt2
ls
merge cmt3 cmt4
ls
read file3 0 10
checkout cmt3
write file3 5 3
lmn
```

```
28 read file3 0 10
```

【样例 2 输出】

```
...ab.....
2 file1 file2
1 file2 file2
...ab.....
2 file1 file3
1 file2 file2
3 file1 file3
..ijk.....
..ijklmn..
```

【样例 2 解释】

此样例中的提交和内容关系如下图所示。

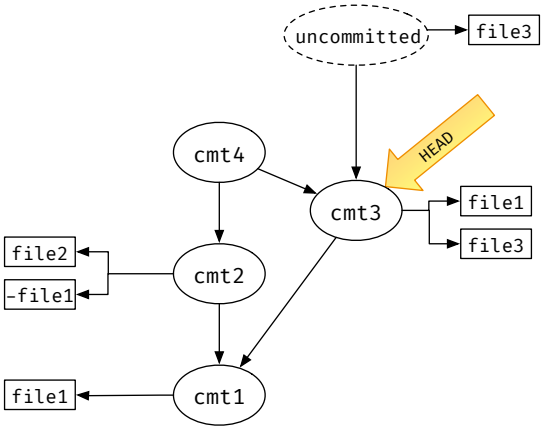


图 4: 样例 2 中的提交关系与内容

输出的第 1 行，对应输入第 7 行的 `read` 命令，其读取到了输入第 2 行中写入的数据；

输出的第 2 行，对应输入第 8 行的 `ls` 命令，此时 `file2` 文件在暂存区中，`file1` 文件在 `cmt1` 中；

输出的第 3 行，对应输入第 11 行的 `ls` 命令，此时刚删除了 `file1` 文件并进行了提交，只有 `file2` 是可以读取的；

输出的第 4 行，对应输入第 13 行的 `read` 命令，此时已经切换到了 `cmt1` 上，因此可以成功读取出 `cmt1` 中保存的 `file1` 的内容；

输出的第 5 行，对应输入第 19 行的 `ls` 命令，此时刚刚创建完 `cmt3`，可以读取的文件包括 `file1` 和 `file3`；

输出的第 6 行，对应输入第 21 行的 `ls` 命令，此时切换回了 `cmt2`，可以读取的文件只有 `file2`；

输出的第 7 行，对应输入第 23 行的 `ls` 命令，此时刚进行完 `merge` 操作，可以读取到的文件包括 `file1`、`file2` 和 `file3`；

输出的第 8 行，对应输入第 24 行的 `read` 命令，此时刚进行完 `merge` 操作，读取到的 `file3` 的内容为 `cmt3` 中保存的 `file3`；

输出的第 9 行，对应输入第 28 行的 `read` 命令，此时可以在暂存区中读取到 `file3` 的内容。此时 `file3` 文件的内容由 `cmt3` 中 `file3` 文件的内容，加上输入第 26 行的写入操作共同构成。

### 【样例 3】

见题目目录下的 ***3.in*** 与 ***3.ans***。在此样例中，最后得到的提交关系和每个提交中的内容与题目开头的示意图相同。

### 【子任务】

本题目中，测试共包含 12 个测试点，每个测试点**包含的命令类型**、**测试规模**、**分值**如下表所示：

| 测试点 | 包含命令类型 | 规模 | 分值 |
|---|---|---|---|
| 1 | read + write | 小规模 | 10 |
| 2 | read + write | 大规模 | 10 |
| 3 | read + write + unlink + ls | 小规模 | 10 |
| 4 | read + write + unlink + ls | 大规模 | 10 |
| 5 | read + write + unlink + ls + commit | 小规模 | 10 |
| 6 | read + write + unlink + ls + commit | 大规模 | 10 |
| 7 | read + write + unlink + ls + commit + checkout | 小规模 | 8 |
| 8 | read + write + unlink + ls + commit + checkout | 大规模 | 8 |
| 9 | read + write + unlink + ls + commit + checkout + merge | 小规模 | 8 |
| 10 | read + write + unlink + ls + commit + checkout + merge | 大规模 | 8 |
| 11 | read + write + unlink + ls + commit + checkout + merge | 超大规模 | 4 |
| 12 | read + write + unlink + ls + commit + checkout + merge | 超大规模 | 4 |

其中**测试规模**表示题目中所涉及到的各项参数的**上限**，具体如下：

| 数据规模 | 小规模 | 大规模 | 超大规模 |
|---|---|---|---|
| 最大文件大小 | 1 KiB | 256 KiB | 2 MiB |
| 总文件数量 | 1000 | 5000 | 5000 |
| 总命令数量 | 1000 | 10000 | 20000 |
| 读写命令中的长度 | 100 | 100 | 100 |
| 总提交个数 | 100 | 2000 | 5000 |
| 所有文件的总大小 | 100 MiB | 1 GiB | 1.5 GiB |

表格中**总文件数量**为 `write` 命令中所出现过的不同文件名的个数。表格中**所有文件的总大小**为整个系统中所有文件的大小之和。如在样例 2 中，总共包括 6 个文件（其中包括一个暗文件），所有文件的总大小为这 6 个文件的大小之和。

## 【提示】

这里是一些非常温馨的提示：

1. 高级命令使得文件操作比较复杂。建议先实现最基本的文件读写，随后通过迭代的方法逐步增加高级功能。
2. 本题目的部分测试点规模较大，在选择编程语言和输入输出方式时请考虑程序运行和输入输出的效率。

# 垃圾回收器（GC）

## 【题目背景】

垃圾回收器（garbage collector, GC）是内存管理中的重要模块，在面向对象的语言运行时（如 Java 虚拟机）中，垃圾回收器会自动找出应用不再使用的对象，释放内存，供应用在未来使用。一种典型的垃圾回收算法是基于追踪（tracing）的，其思路如下图所示：

1. 首先，垃圾回收会找到一些已知的存活对象（例如存放重要全局变量的对象）。这些对象称为**根对象**，是垃圾回收的起点，存放它们引用的数据结构称为**根表**。
2. 之后，垃圾回收会从根对象（图中深灰色）开始，通过对象之间的引用关系找到所有存活对象（图中浅灰色）。为了完成这一步，垃圾回收器需要知道对象中哪些位置存放的是引用。一种方法是提供**类型表**保存所有类型信息，并在每个对象中维护一个指向其类型的引用。通过获取类型信息，垃圾回收器就能找到对象中的引用，从而进一步找到存活对象。
3. 最后，垃圾回收会对存活对象进行整理，通过拷贝使它们紧凑地排列在一起，从而腾出空闲空间供应用使用。

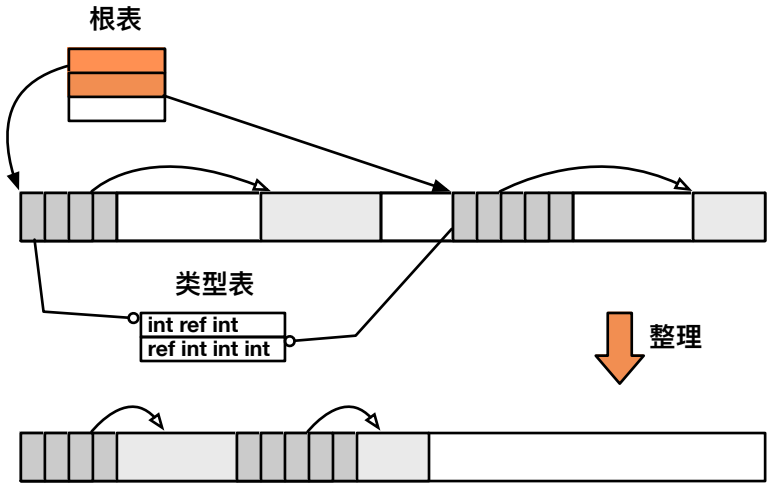


图 5: 垃圾回收概览

## 【题目描述】

在本题中，你需要实现一个简单的垃圾回收器 ***SimpleGC***，对内存资源进行回收。在实际系统中，垃圾回收的设计是与内存分配器紧密相关的。为了使问题简化，本题省略了内存分配过程，而是采用给出**堆镜像**（heap image）的方法。堆镜像会提供垃圾回收前的内存数据，包含了根表、类型表、堆内数据等信息，而你的 ***SimpleGC*** 需要对

这些内存数据进行整理。堆镜像将以字节数组的形式给出，其大小固定为 64MB，格式如下（也可参考后面的样例说明）：

- 数组里的第一个数字为根表包含的根对象数量 R （1<=R<=1024），占用内存为 4 个字节。**注意**：对于这种存放在多个字节中的数字，由于堆镜像是以字节为单位给出数据，我们需要将各个字节的数据组合在一起，从而计算出存放的数据。假设我们有 n 个字节 $a_0$、$a_1$、...、$a_{n-1}$，那么转换出的数据应该为：
  - $a = a_0 + a_1 * 256^1 + a_2 * 256^2 + ... + a_{n-1} * 256^{n-2}$

  后续 R 个数字表示对于不同根对象的引用（引用保证不会重复），每个数字占用内存也为 4 个字节。
- 第 R+2 个数字为类型表包含的对象类型数量 T （1<=T<=1024），占用内存为 4 个字节。后续的数字是对这些类型的描述。对于某种类型，第一个数字 K（1<=K<=1024）表示类型中包含的成员变量数量（占用 4 个字节），之后 K 个数字每个都表示一个成员变量的类型，每个数字占用 1 个字节。为了简化，成员变量的类型只可能是以下四种中的一种：
  - 字符类型（char），对应类型为 0，占用内存为 1 个字节；
  - 短整数类型（short），对应类型为 1，占用内存为 2 个字节；
  - 整数类型（int），对应类型为 2，占用内存为 4 个字节；
  - 引用类型（reference），对应类型为 3，占用内存为 4 个字节。每个引用类型会指向堆内的一个对象（即保存该对象在堆内的地址）。如果其值为 **1**，则表示该引用为空。
- 数组的最后一部分记录堆大小和堆内数据。第一个数字 H 表示堆内数据包含的字节数（8<=H<=16M），占用 4 个字节；后面 H 个数字表示堆内每个字节里存放的数据。堆内数据的起始地址为 0。需要注意的是，堆内保存的数据全部以**对象**为单位，字符、短整数、整数、引用类型都只能作为成员变量包含在对象之中，不能独立存在。由于内存中支持不同类型的数据，因此本题规定，在堆中，不同类型的起始地址应当遵循一定的对齐要求：
  - 字符类型没有对齐要求，可以出现在堆的任意位置；
  - 短整数类型遵循 2 对齐，它的起始地址要能被 2 整除；
  - 整数类型和引用类型遵循 4 对齐，它的起始地址要能被 4 整除；
  - 对于每个对象来说，其起始地址要能被 4 整除；
  - 对于任何一种类型，如果其起始地址不能满足对齐要求，那么应该从当前的内存地址开始填入若干个 0，直到满足对齐要求。

下面的例子展示了堆内一个对象的结构。对该结构的说明如下：

- 对于每个对象，前四个字节会存储它的对应类型。在该例子中，前四个字节存储的类型为 2 （$2 + 0 * 256 + 0 * 256^2 + 0 * 256^3$），因此可以从类型表的第二个条目中找到对于该类型的描述。
- 该对象的第一个成员变量为字符类型，第二个为短整数类型。为了满足 2 对齐需求，堆镜像会填入一个 0（图中红色字体），使短整数的起始地址变为 6。类似地，短整数的值为 513（$1 + 2 * 256$）
- 该对象的第三个成员变量为字符类型，第四个为整数类型。为了满足 4 对齐需求，内存镜像会填入三个 0，使整数的起始地址变为 12。类似地，整数的值为 18088460。

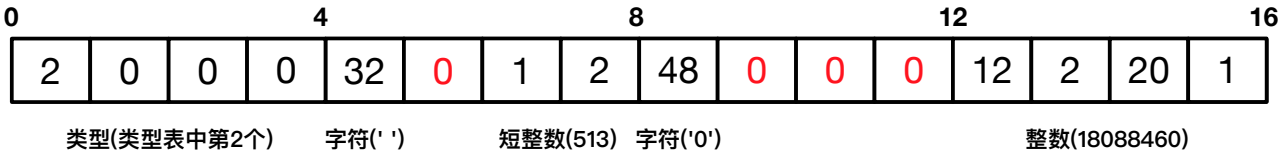


图 6: 对象结构举例

在垃圾回收过程中，***SimpleGC*** 只能进行以下三类的内存整理：

- 在堆内数据中，拷贝存活对象，腾出更多可用的空间；
- 在类型表中，修改类型的结构，使其占用空间更小，同时修改堆内数据中对应类型对象的结构，以保证正确性；需要注意的是，对于一个类型中同种类的两个成员，它们的顺序在垃圾回收之后不能发生变化。举例来说，对于上面的对象，可能的结构调整方式如下。通过将第三个成员变量前移，可以使其他变量都能直接对齐，从而省去了四个字节的空间。不过，由于第三个变量和第一个变量类型相同（字符类型），**它们的位置不能对调**。

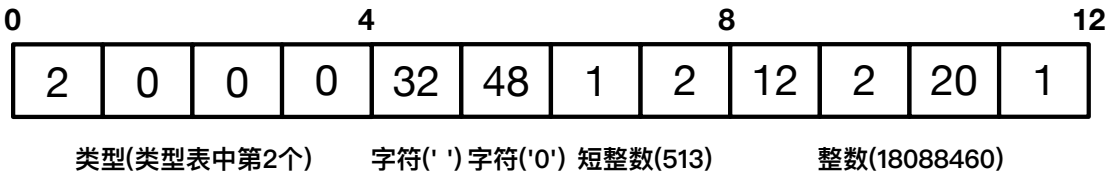


图 7: 调整结构后的对象

- 删除类型表中没有使用的条目。类似地，**请不要改变已有的条目在堆镜像中的相对顺序**。

**注意**：你只能进行这三类优化，其他类型的优化（如对数据进行压缩）无法通过正确性检测。

## 【答题接口】

本题提供 C/C++/Java/Python 四种语言的解题框架。下面以 C 版本为例进行说明，C++、Java、Python 版本要求类似。

在 gc.c 中传入了一个 ***mem*** 数组，它保存了当前的堆镜像（包括根表、类型表、内存数据）。你需要实现 gc 这一函数，整理该数组：

```
1 void gc(unsigned char *mem);
```

为实现该函数，你可以实现任意数量的辅助函数。

当该函数调用返回之后，评测程序 ***verifier*** 会对输出的堆镜像进行检查，并给出成绩。

## 【输入格式】

本题的测试程序会运行 20 个测试点，每个测试点会从文件中读入堆镜像，保存在数组中。之后，测试程序会调用你实现的 gc 函数对数组进行整理，并进行正确性检验以及相应性能指标的统计。为了帮助选手在本地进行调试，我们提供了和最终评测数据分布相同的 20 个测试点。选手可以在网页上下载这些测试点。

下面展示了两个测试点文件中的输入样例及它们对应的堆结构：

## 【样例 1 输入】

```
1 1 0 0 0 16 0 0 0 2 0 0 0 4 0 0 0 0 1 2 3 1 0 0 0 2 32 0 0 0 1 0
    0 0 1 0 0 0 1 0 0 0 2 0 0 0 0 0 0 0 12 0 12 23 12 9 3 7 0 0
    0 0
```

## 【样例 1 解释】

如下图所示，该堆共包含两种自定义类型，一个根对象，三个对象。

堆整理后，只包含两个存活对象，回收了 8 个字节的内存。

如果以字符为单位输出整理后数组的数据，会得到以下的结果（你也可以在 gdb 中使用 x/uc 的方式查看）：

```
1 1 0 0 0 0 0 0 2 0 0 0 4 0 0 0 0 1 2 3 1 0 0 0 2 24 0 0 0 0 0
   0 0 12 0 12 23 12 9 3 7 16 0 0 0 1 0 0 0 1 0 0 0
```

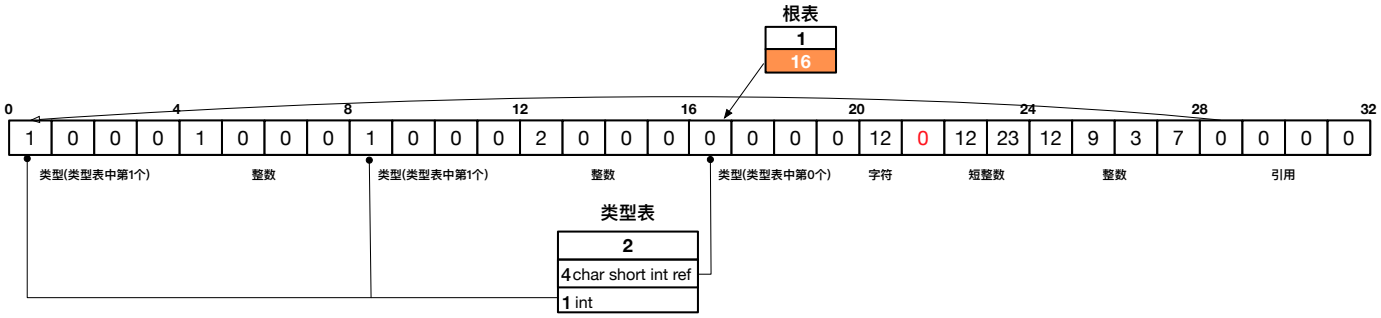


图 8: 样例 1 输入的堆结构

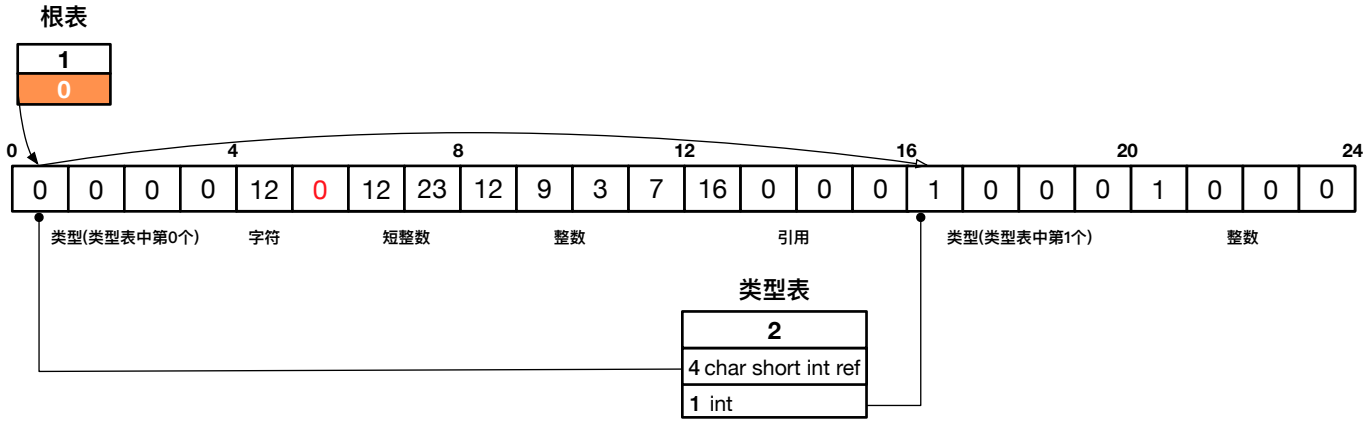


图 9: 样例 1 GC 后的堆结构

**【样例 2 输入】**

```
1 0 0 0 0 0 0 0 2 0 0 0 3 0 0 0 0 3 0 1 0 0 0 2 16 0 0 0 0 0 0
  0 32 0 0 0 1 0 0 0 65 0 0 0
```

**【样例 2 解释】**

该堆只包含一个对象，其引用为空。

垃圾回收没有改变存活对象的数量，但调整了类型的结构，并删除了没有使用的类型表条目，节省了 9 个字节（如下图）。

如果以字符为单位输出整理后的数据，会得到以下的结果：

```
1 0 0 0 0 0 0 0 1 0 0 0 3 0 0 0 0 0 3 12 0 0 0 0 0 0 0 32 65 0
  0 1 0 0 0
```

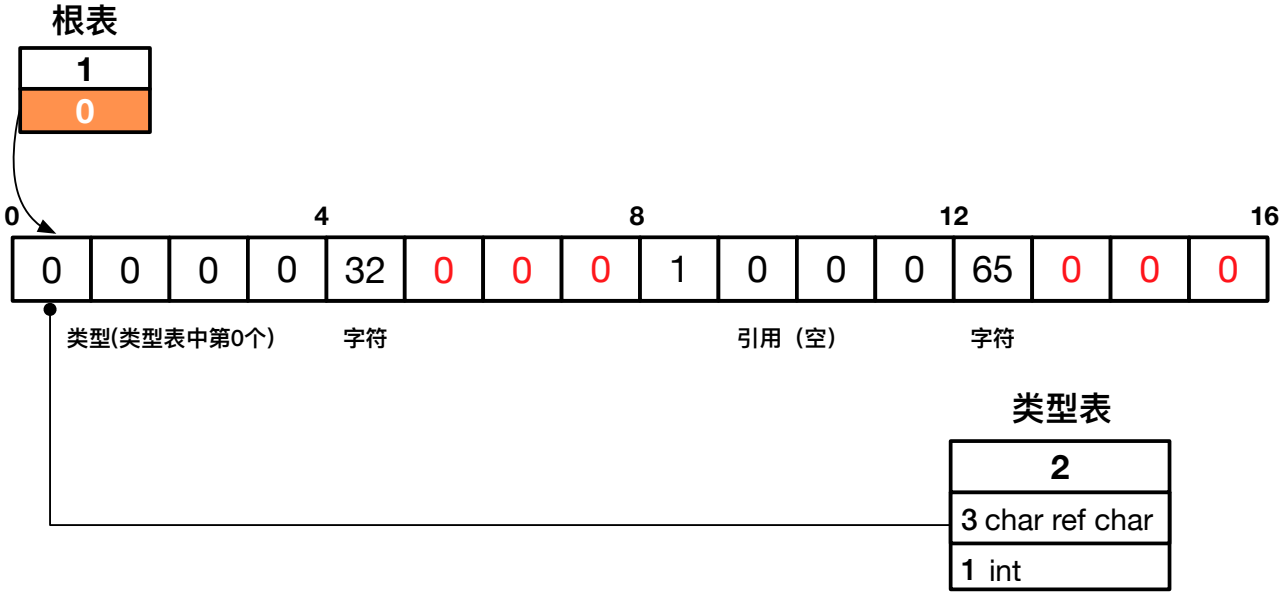


图 10: 样例 2 输入的堆结构

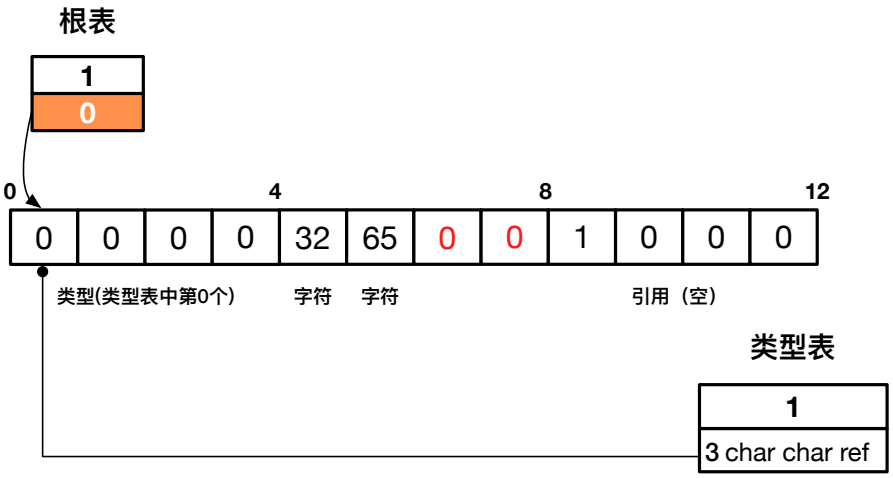


图 11: 样例 2 GC 后的堆结构

## 【输出格式】

我们提供了测试程序 tester.c/cpp/java/py 及验证程序 verifier。当你实现完 GC 及相关辅助函数之后，即可编译并运行验证程序，进而输出相应的测试结果，其中包括正确性、回收内存效率以及耗费时间等指标信息。对于某个测试点，如果你的程序未能通过正确性检验，则并不会输出耗费时间等信息。其中回收内存率的输出为 (0, 1) 之间的小数，耗费时间的输出为以秒为单位的小数，两者都保留到小数点后 7 位。需要注意的是，如果你的某个测试点回收内存效率低于 0.1，该测试点也会被视为不正确。对于这种情况，测试结果只会输出回收效率，并不会输出耗费时间。

- **回收内存效率**：回收内存效率为调用你的 GC 之后堆镜像的内存总大小与调用前的内存总大小之比。例如在上面的样例 1 中，回收前共占用 61 字节，回收后共

占用 53 字节，则回收内存效率为 0.1311480 ((61-53)/61)。

- **耗费时间**：耗费时间只包括调用选手实现的 GC 函数的时间。需要注意，这里的耗费时间与评测网站上显示的时间限制（30s）不同，因为时间限制包括了框架和验证程序的运行时间，但耗费时间不包括。对于每个测试点，我们保证框架和验证程序的运行时间不超过 10s，因此你需要使你的 GC 运行总时长不超过 20s 才能不超过时间限制。

对于多个测试点，可能的测试运行结果如下：

```
yes    0.1311480      0.0123190        1.in
no     0.0844524              -        2.in
no             -              -        3.in
...
```

上述输出表示，你的 GC 可以通过 1.in 的测试，其中回收效率为 0.131148，耗费时间为 0.0123190s。之后，你的 GC 虽然能通过 2.in 的正确性检测，但它的内存回收效率过低，因此也被判为不正确。最后，你的 GC 不能通过 3.in 的正确性检测。

为了方便调试，我们在本地也提供了相应的验证程序 verifier。选手可以运行

```
./verifier <case_number> <result> <info>
```

该程序接收三个参数：测试点号（case number）、结果文件（result）、详细信息文件（info）。verifier 会根据测试点号操作相应的文件，并在检查完正确性后，将该测试点的得分输出到 result 中，并将上述的指标信息（包括正确性、回收效率、耗费时间等）输出到 info 中。例如，选手可以调用

```
./verifier 1 result.out info.out
```

此时，verifier 会找到 1.in 读取 GC 前的堆镜像，与 1.out 中使用选手 GC 以后的堆镜像进行对比以确认正确性和回收效率。之后，verifier 会将分数和指标信息分别输出到 result.out 和 info.out。请注意，1.in 和 1.out 文件需要都放在**当前目录**下。

## 【评分方式】

本题的分数分为两部分：正确性分数和性能分数。

正确性分数满分为 20 分，包括 20 个测试点，每个测试点 1 分。验证程序会从选手输出的堆镜像中根表内的引用出发，遍历所有的存活对象，并与输入数据进行对比。对于每个测试点，如果选手的程序能够在给定的时间、空间限制下输出堆镜像并且通过正确性测试，则得 1 分，否则得 0 分。**注意，选手的程序至少需要通过前两个测试点，否则整题得分为 0 分。**

性能分数满分为 80 分，只考虑 5-20 这 16 个测试点，每个测试点的满分为 5 分。若正确性分数为 0 分，那么性能分数也为 0 分；否则每个测试点的得分将综合考虑**回收内存效率**和**耗费时间**。

我们为每个测试点提供了作为基线的回收内存效率和耗费时间。对于第 i 个测试点，设 ei 为选手程序的内存回收效率，ti 为选手程序的耗费时间，Ei 为基线回收内存效率值，Ti 为基线耗费时间值，并取：

$ai = ei/Ei$

$bi = Ti/ti$

那么选手的得分 si 为：

$si = 5 * (0.7ai + 0.3bi)$

选手的性能分数为各测试点性能得分之和，总分为性能分数和正确性分数之和，共计满分 100 分。

## 【提示】

- 测试环境配置为双核机器，因此选手可以使用多线程来加速程序的性能。