内存管理

> 主要考察析构函数

由于 libc 原生的内存管理 API 的种种问题（例如，内存泄露和 double-free），某些数据库管理系统基于这些 API 实现了新的动态内存管理机制。具体地，这些实现中引入了一个称为“memory context”的概念（后文简称为“MC”），动态内存管理不再直接使用 `malloc` / `free` 等函数，而是调用某个 MC 对象提供的 API 进行。这些 MC 对象被组织为树状结构，如图 1 所示。这样，使用者不再需要为之前分配的每个内存块调用 `free` 等函数，而是等到相应的 MC 对象被销毁时统一地归还其下的动态内存。同时，由于树状结构的存在，在销毁较为高层的 MC 对象时，处于低层次的 MC 对象也同样会被销毁。这种机制极大降低了管理动态内存的心智负担，更好地避免了内存泄露和 double-free 等问题。

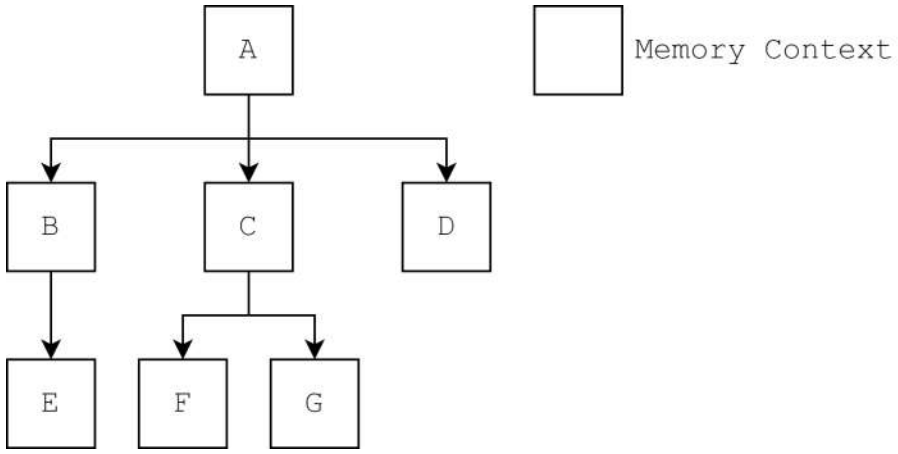

图 1：MC 的树状结构

我们提供了骨架代码，请填充其实现。需要实现如下的功能：

- 维护 MC 对象之间的父子关系
- 给定一个 ID，分配一块内存并将二者关联起来
- 在析构函数中进行上述的销毁工作

输入描述

本题不需要处理输入。

输出描述

销毁某个 ID 为 `X` 的内存块时，输出

```
Chunk X freed.
```

对于一个 MC 下的内存块，按照**分配顺序的逆序**（即，后进先出）的顺序进行销毁。

如果一个 MC 有多个子 MC，那么按照创建这些子 MC 的顺序进行销毁。

示例

示例 1

MC 结构和操作序列

图 2：示例 1 的 MC 结构

操作序列为：

1. A.alloc("1")
2. A.alloc("2")
3. A.alloc("3")

输出

```
Chunk 3 freed.
Chunk 2 freed.
Chunk 1 freed.
```

示例 2

MC 结构和操作序列

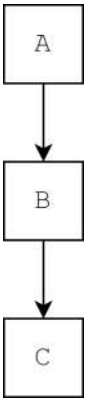

图 3：示例 2 的 MC 结构

操作序列为:

1. a.alloc("1")
2. a.alloc("2")
3. a.alloc("3")
4. b.alloc("1/1")
5. b.alloc("1/2")
6. b.alloc("1/3")
7. c.alloc("1/1/1")
8. c.alloc("1/1/2")
9. c.alloc("1/1/3")

输出

```
Chunk 1/1/3 freed.
Chunk 1/1/2 freed.
Chunk 1/1/1 freed.
Chunk 1/3 freed.
Chunk 1/2 freed.
Chunk 1/1 freed.
Chunk 3 freed.
Chunk 2 freed.
Chunk 1 freed.
```

示例 3

MC 结构和操作序列

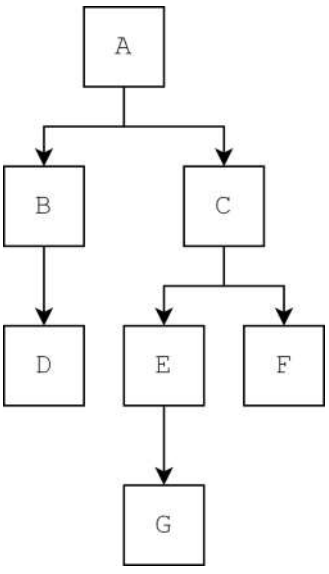

图 4：示例 3 的 MC 结构

操作序列：

1. a.alloc("1")
2. a.alloc("2")
3. a.alloc("3")
4. b.alloc("1/1")
5. c.alloc("1/2")
6. d.alloc("1/1/1")
7. d.alloc("1/1/2")
8. g.alloc("1/2/1/1")

输出

```
Chunk 1/1/2 freed.
Chunk 1/1/1 freed.
Chunk 1/1 freed.
Chunk 1/2/1/1 freed.
Chunk 1/2 freed.
Chunk 3 freed.
Chunk 2 freed.
Chunk 1 freed.
```