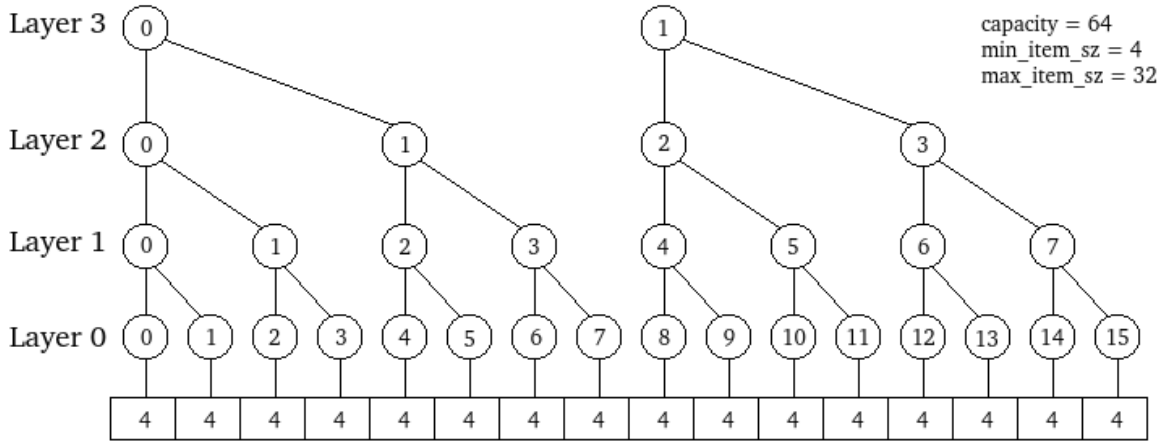
Buddy Allocator, 但是 C++

题目描述

ACM 班的操作系统课程大作业是用 Rust 写一个微内核. 其中需要实现一个内存分配器来为操作系统管理内存, Conless 决定使用 Buddy Allocator 来实现. Buddy Allocator 是一种内存分配算法, 它将内存分割成大小为 2 的幂次方的块, 来高效应对不同的内存申请/释放需求.

Buddy Allocator 的原理是这样的:



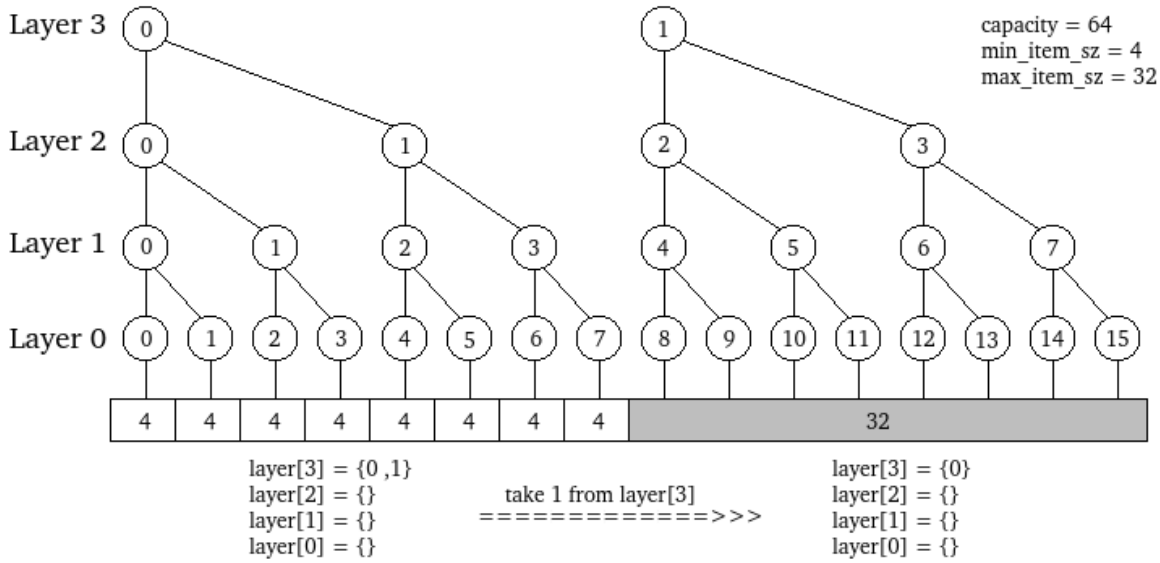
假设总共有 64 个 byte 的可用内存空间 (例如 0x00 - 0x3f), n内存的最小分配单元为 4 byte, 那么 buddy allocator 将这 64 个 byte 的内存空间分割成大小为 4, 8, 16 和 32 的块, 产生

- 16 个 4 byte 的块, 分别对应内存地址 0x00 - 0x03, 0x04 - 0x07, ..., 0x3c - 0x3f
- 8 个 8 byte 的块, 分别对应内存地址 0x00 - 0x07, 0x08 - 0x0f, ..., 0x38 - 0x3f, 每个块 *包含* 两个 4 byte 的块
- 4 个 16 byte 的块, 分别对应内存地址 0x00 - 0x0f, 0x10 - 0x1f, 0x20 - 0x2f, 0x30 - 0x3f, 每个块 *包含* 2 个 8 byte 的块
- 2 个 32 byte 的块, 分别对应内存地址 0x00 - 0x1f, 0x20 - 0x3f, 每个块 *包含* 2 个 16 byte 的块

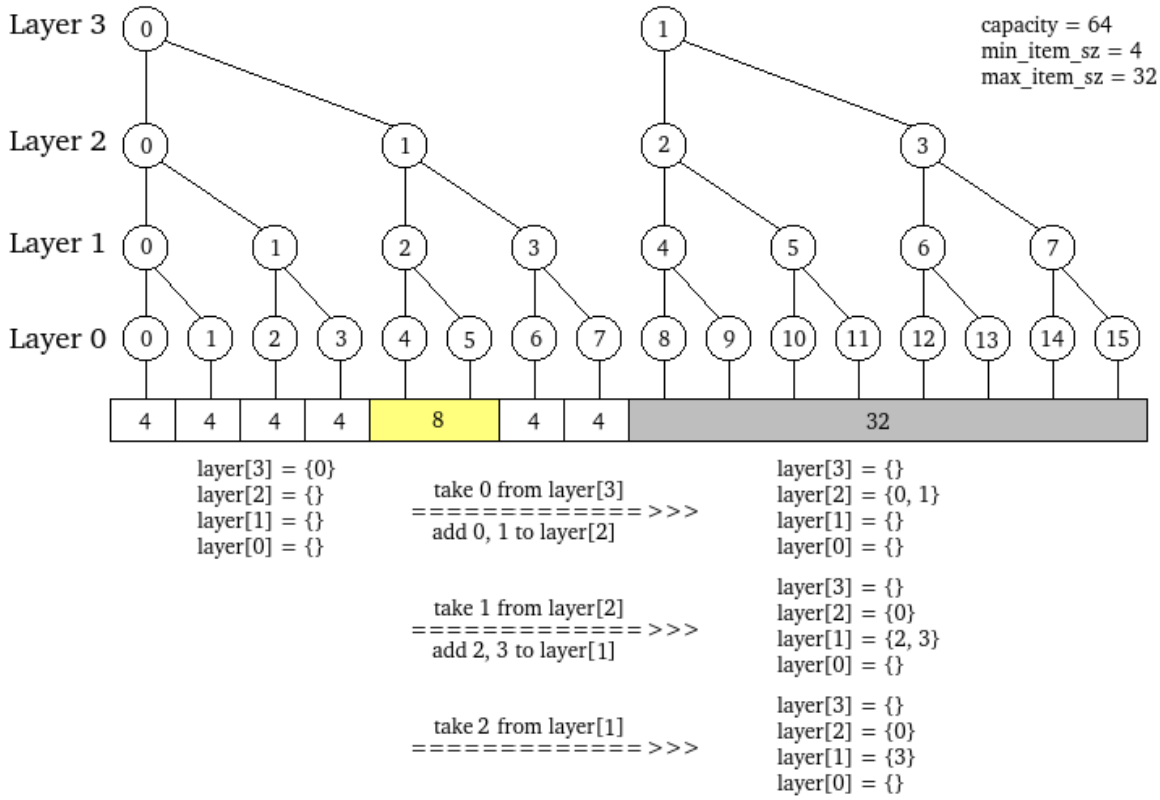
如上图所示.

在初始状态下, 只有最上层 (layer 3) 的两个块为空闲, 下层块由上层节点块分裂产生 (具体案例见下), 也可以合并重新产生上层节点. 下面是一些连续的操作示意:

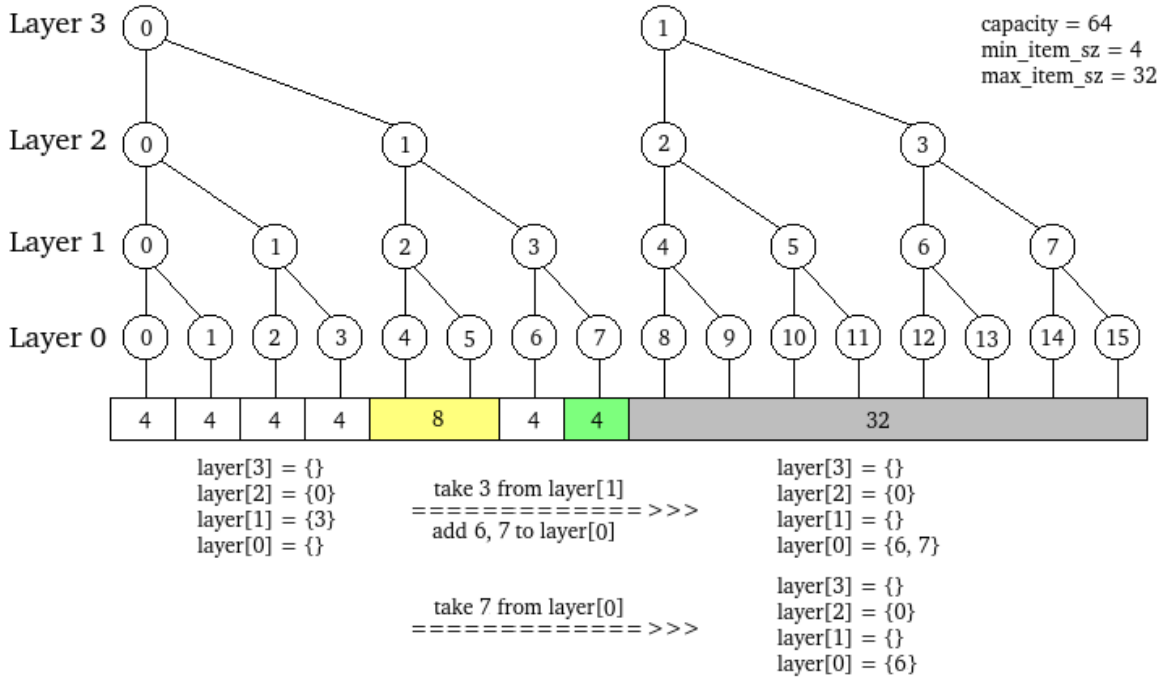
1. 若在内存地址 0x20 处申请 32 byte 内存, allocator 将检查 32 byte 对应的 layer 3 是否在 0x20 (对应下标 1) 处有空闲块, 发现有, 则将其分配给申请者. 见下图.



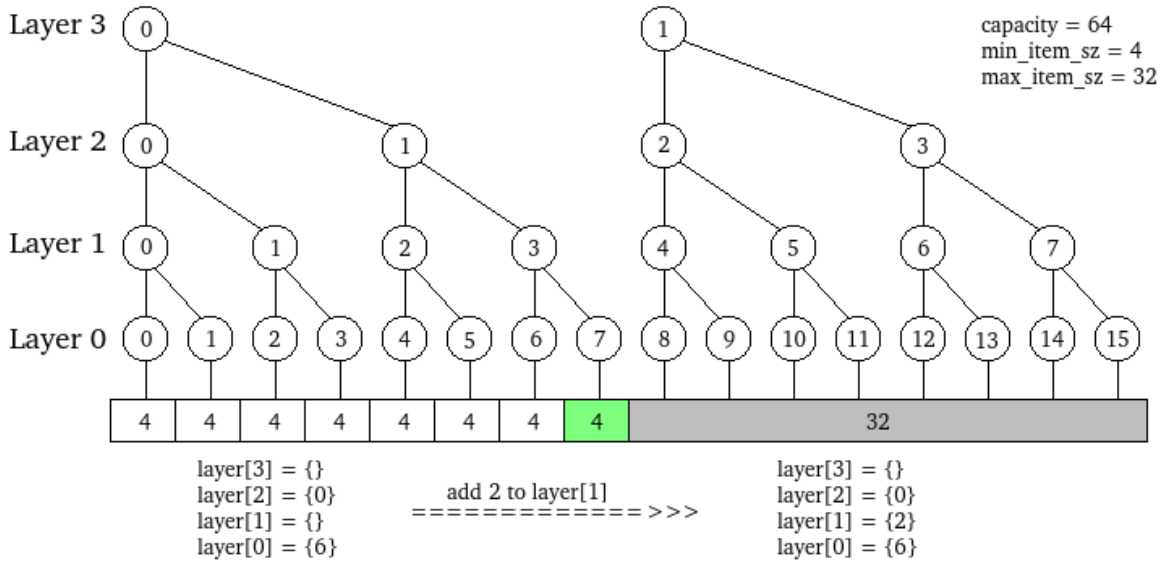
2. 若在内存地址 0x10 处申请 8 byte 内存, allocator 将检查 8 byte 对应的 layer 1 是否在 0x10 (对应下标 2) 处有空闲块. **发现没有, 试图向上层申请分裂**. 于是 layer 2 收到分裂下标 1 的申请, 发现此处仍然没有空闲块, 再向上层申请分裂. 最终 layer 3 收到分裂下标 0 的申请, 发现此处有空闲块, 则将其分裂为 layer 2 的两个块 0 和 1; 随后 layer 2 再将块 1 分裂为 layer 1 的两个块 2 和 3; 最后 layer 1 将块 2 分配给申请者. 见下图.



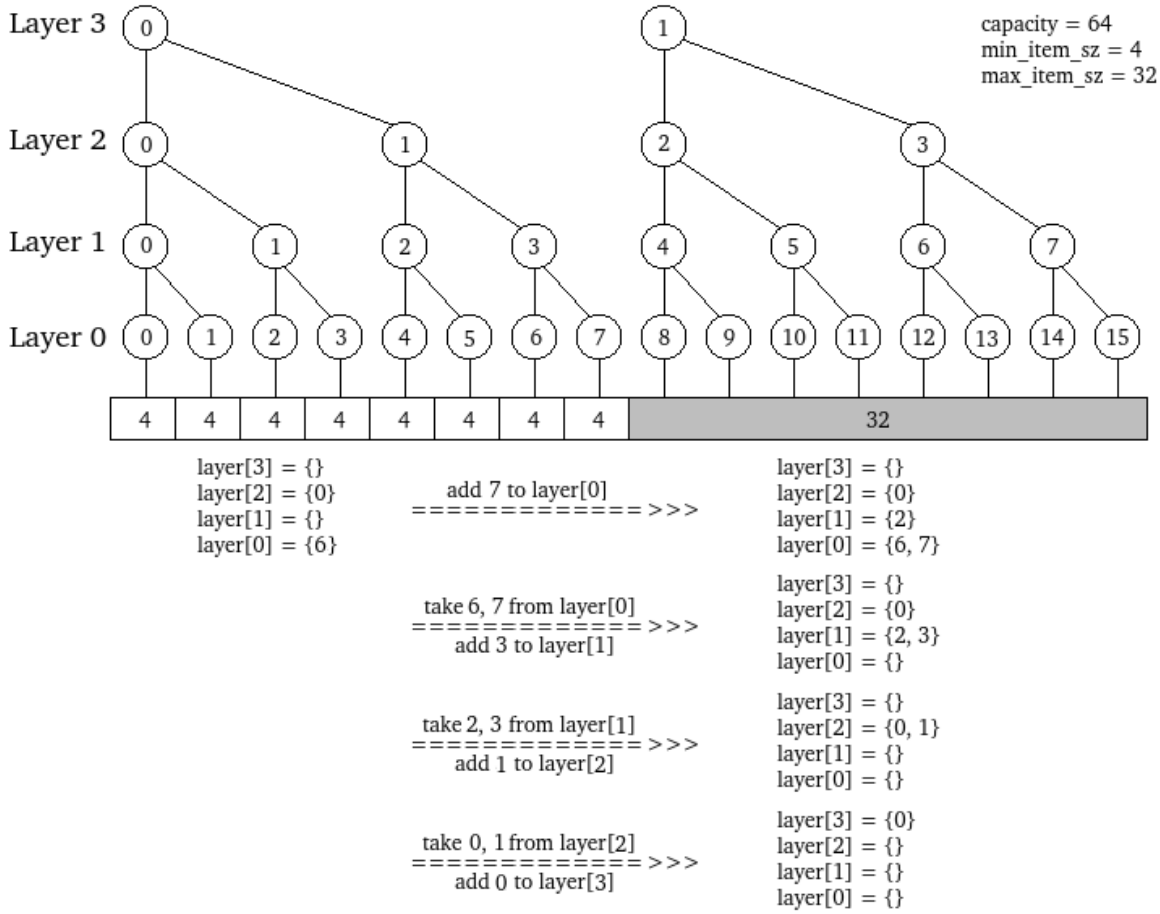
3. 若在内存地址 0x1c 处申请 4 byte 内存, allocator 将检查 4 byte 对应的 layer 0 是否在 0x1c (对应下标 7) 处有空闲块. 发现没有, 试图向上层申请分裂. 于是 layer 1 收到分裂下标 3 的申请, 发现此处有空闲块, 则将其分裂为 layer 0 的两个块 6 和 7; 最后 layer 0 将块 7 分配给申请者. 见下图.



4. 若释放第二步操作申请的 0x10 处的 8 byte 内存, allocator 将向 layer 1 中加入块 2, 并查看块 2 的兄弟节点块 3 是否也在 layer 1 中, 发现不在 (即不能合并), 结束释放过程. 见下图.



5. 若释放第三步操作申请的 0x1c 处的 4 byte 内存, allocator 将向 layer 0 中加入块 7, 并**查看块 7 的兄弟节点块 6 是否也在 layer 0 中, 发现在 (即可以合并), 则将块 6 和 7 合并为 layer 1 的块 3**, 并查看块 3 的父节点块 1 是否也在 layer 1 中, 发现在 (即可以合并), 则将块 1 和 3 合并为 layer 2 的块 1, 并查看块 1 的父节点块 0 是否也在 layer 2 中, 发现在 (即可以合并), 则将块 0 和 1 合并为 layer 3 的块 0, 并结束释放过程. 见下图.



通过这样的分配算法, buddy allocator 可以以较高的效率实现不同幂次大小的内存块的申请和释放.

题目要求

不幸的是, Conless 要去出数据结构机考题, 所以只好把这一艰巨的任务交给你了. 出于仁慈, 他不准备让你们用 Rust 写, 而是用 C++ 实现其中的接口.

请实现一个基于 buddy allocator 算法 (允许修改具体细节) 的内存分配器, 以可用内存大小 ram_size 与最小块大小 min_block_size 初始化, 支持以下三种操作:

- `int malloc(int size)`：申请大小为 $size$ 的内存块 (为了方便评测, 本题要求申请的内存块是以 $size$ 的整数倍为起点中地址最小的可用内存块, 即若 $size = 8$, 当前可用内存为 $0 - 4, 16 - 63$, 则返回 16). 若成功分配, 返回申请的内存块的起始地址, 否则返回 `-1`.
- `int malloc_at(int addr, int size)`：在 $addr$ 处申请大小为 $size$ 的内存块. 若成功分配, 返回 `addr`, 否则返回 `-1`.
- `void free_at(int addr, int size)`：释放 $addr$ 处大小为 $size$ 的内存块. 保证该内存块一定由成功的 `malloc_at` 操作得到.

请按上述要求完成下发文件中的 `src.hpp` 并将其提交至 OnlineJudge.

样例输入/输出

见下发文件中的 `test.cpp`.

数据范围

设总操作数为 Q.

可用内存地址均从 0 开始. 保证 ram_size 与 $size$ 均为 min_block_size 乘以 2 的整数次幂, $addr$ 是 $size$ 的整数倍且小于 ram_size. 除申请已经被申请过的内存区域外, 你不需要考虑任何非法操作 (例如 double free, 访问越界等).

对于 20% 的数据, 保证 Q 与 ram_size 不大于 100.

对于 40% 的数据, 保证 Q 与 ram_size 不大于 5×10^3.

对于 100% 的数据, 保证 Q 与 ram_size 不大于 3×10^5, min_block_size 的取值为 1, 2 或 4.

本题将在 50% 的测试数据上开启内存泄漏检查, 请确保你的程序是内存安全的. 你可以在本地使用 `valgrind` 进行自查.