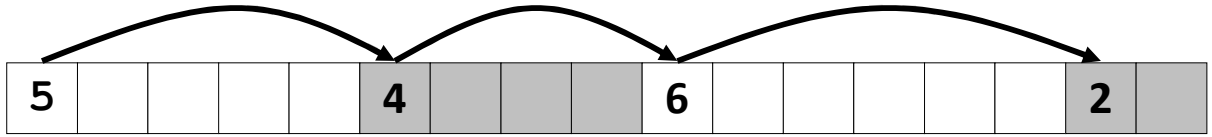
动态内存分配：进阶概念
(Dynamic Memory Allocation)

主要内容

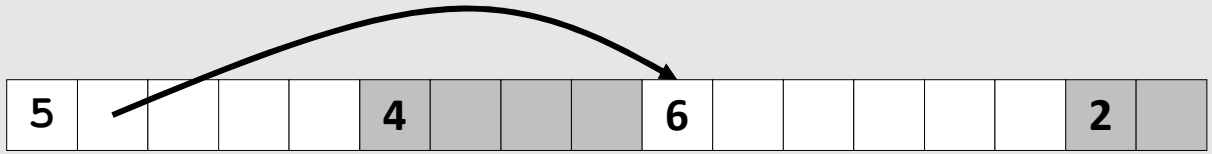
- 显式空闲列表
- 分离空闲列表
- 垃圾回收
- 内存相关的风险和陷阱

追踪空闲块的方法

- 方法 1：隐式链表（Implicit list）使用块的长度信息，把所有块串联起来



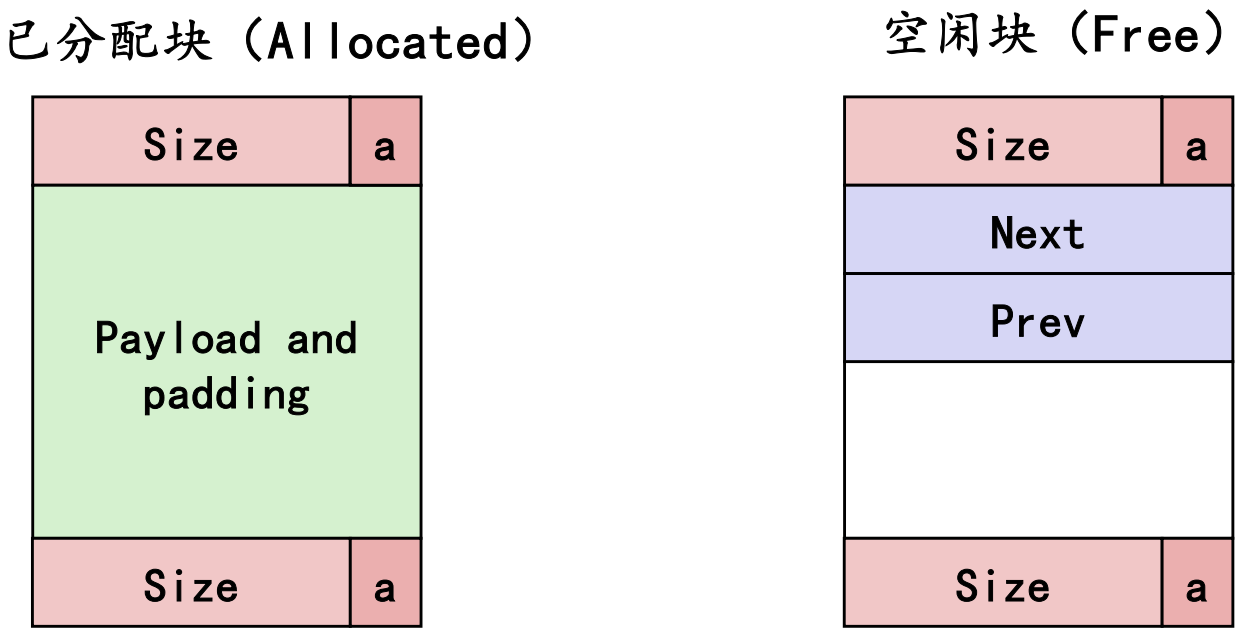
- 方法 2：显式链表（Explicit list）仅在空闲块之间使用指针建立链表



- 方法 3：分离空闲链表（Segregated free list）为不同大小类别维护不同的空闲链表

- 方法 4：按大小排序的块（Blocks sorted by size）
  - 使用平衡树（如红黑树），在每个空闲块里保存指针，并以块的长度作为关键字进行组织

显式空闲链表

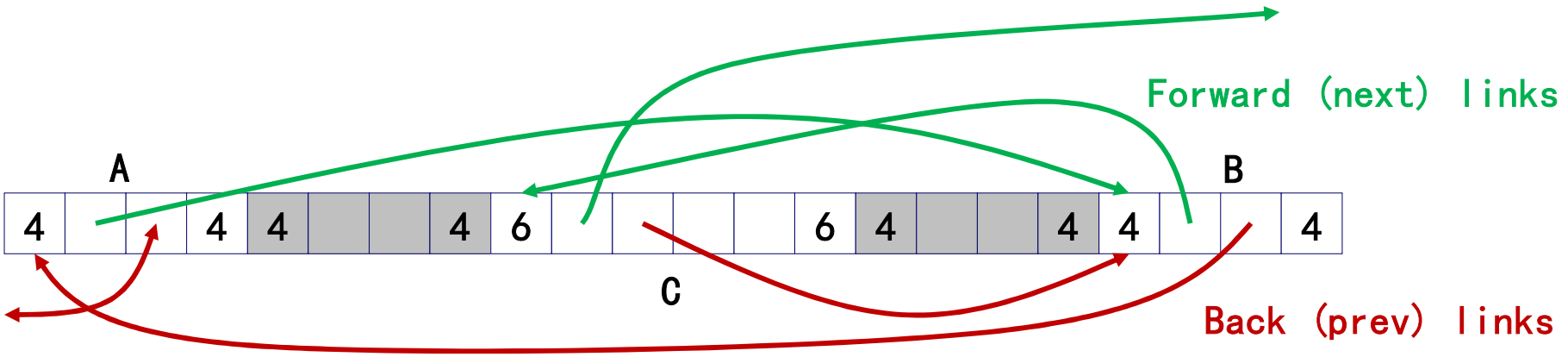


- 我们维护的是空闲块的链表，而不是所有块的链表。
  - “下一个”空闲块的位置可能在堆的任何地方，因此我们必须在块中保存前向和后向指针，而不仅仅是块的大小。
  - 仍然需要使用边界标记（boundary tags），以便在释放时可以合并相邻空闲块。
  - 幸运的是，我们只需要在空闲块里维护这些额外指针，因此可以利用空闲块的有效负载区来存放这些指针，而不会浪费额外的空间。

显式空闲链表（Explicit Free Lists）

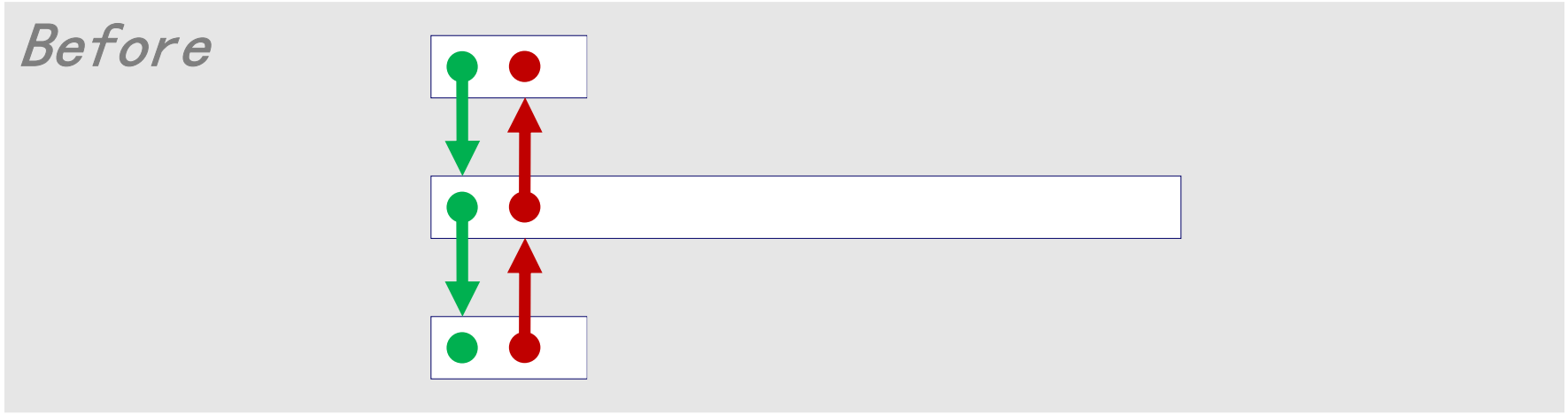
- 逻辑上：

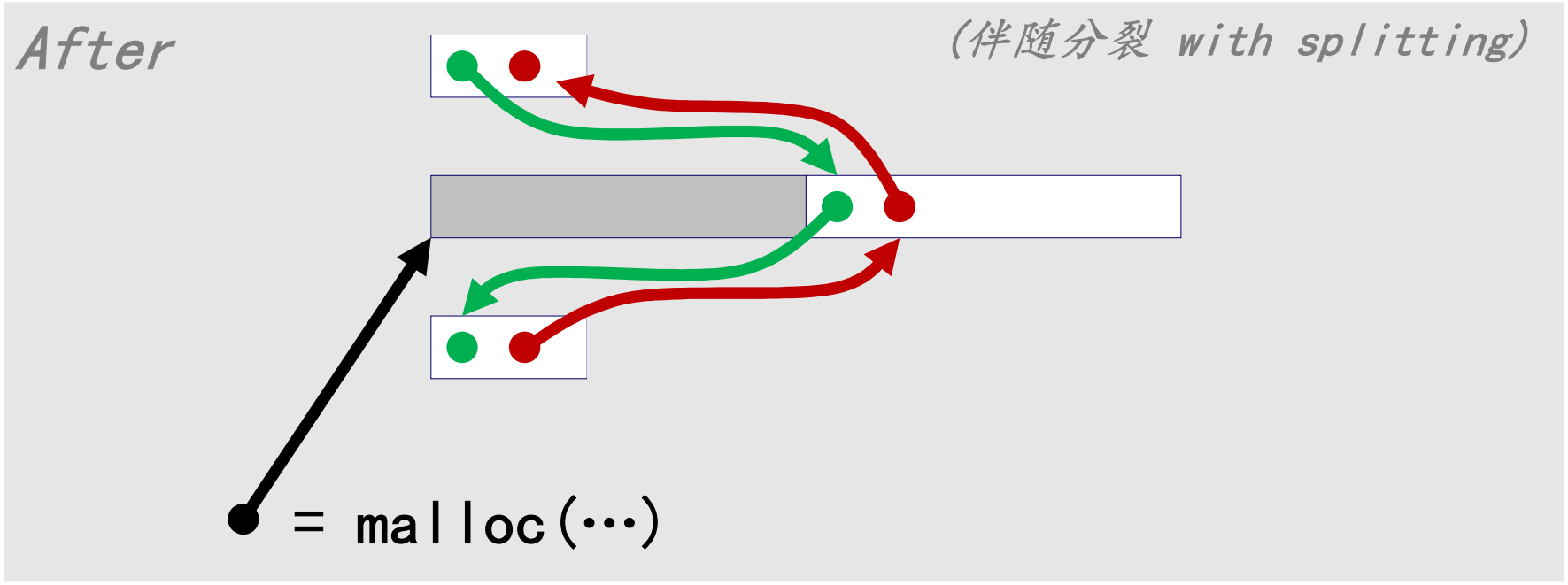
- 物理上：空闲块在堆中的位置可以是任意的，不需要相邻。
  - 前向指针（Forward, next links）用来指向下一个空闲块
  - 后向指针（Back, prev links）用来指向前一个空闲块。



从显式空闲链表中分配内存

概念图



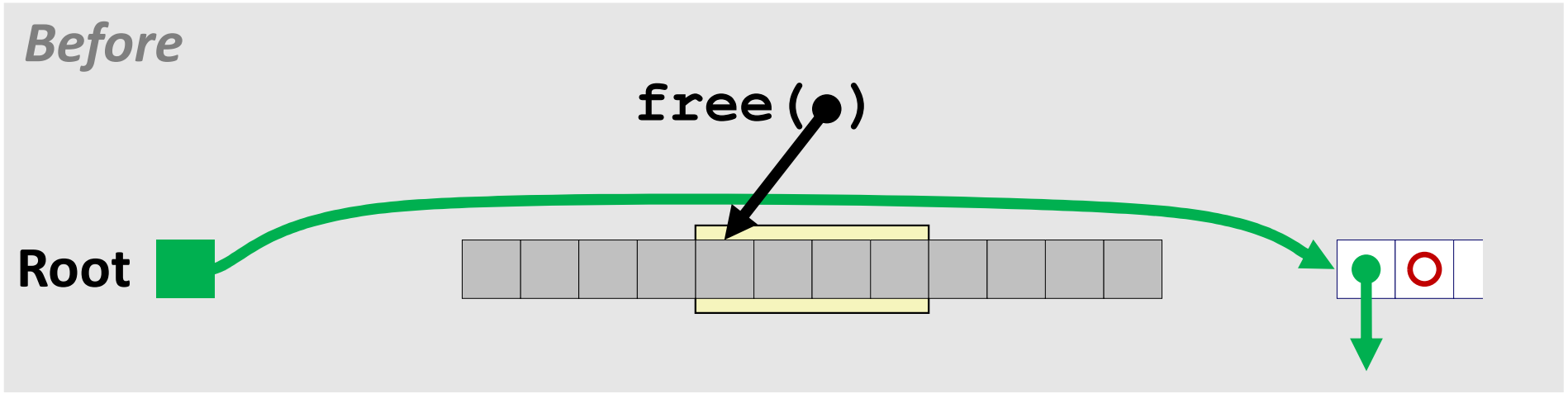


使用显式空闲链表释放内存

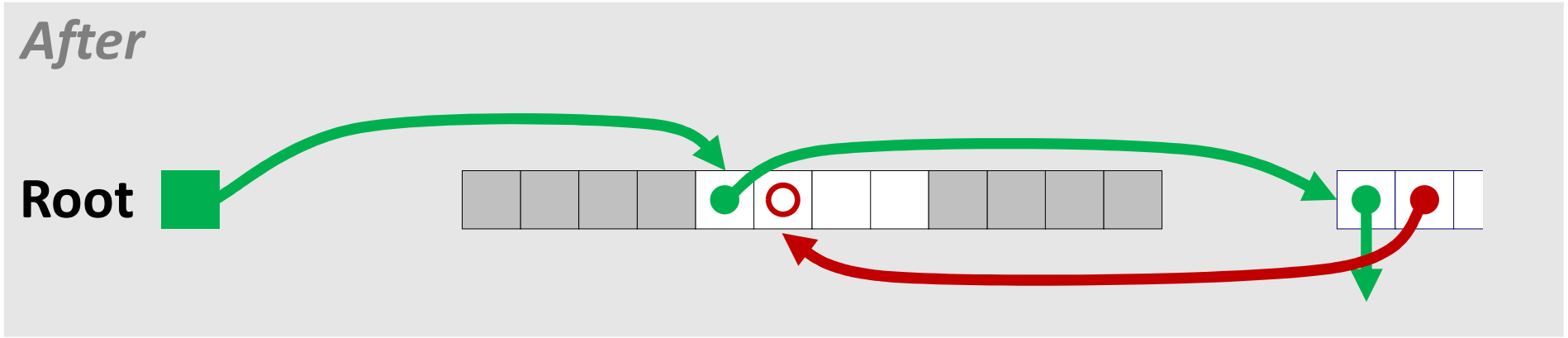
- 插入策略（Insertion policy）：在空闲链表中，新的空闲块应该放在哪里？
- LIFO（后进先出）策略
  - 将释放的块插入到空闲链表的开头
  - 优点（Pro）：简单，耗时为常数时间
  - 缺点（Con）：研究表明，这种方式的碎片化情况通常比按地址顺序更严重
- 按地址顺序策略（Address-ordered policy）
  - 将释放的块按地址顺序插入空闲链表，始终满足：
    addr(prev) < addr(curr) < addr(next)
  - 缺点（Con）：需要进行查找操作
  - 优点（Pro）：研究表明，这种方式的碎片化程度比 LIFO 更低

使用 LIFO 策略释放（案例 1）

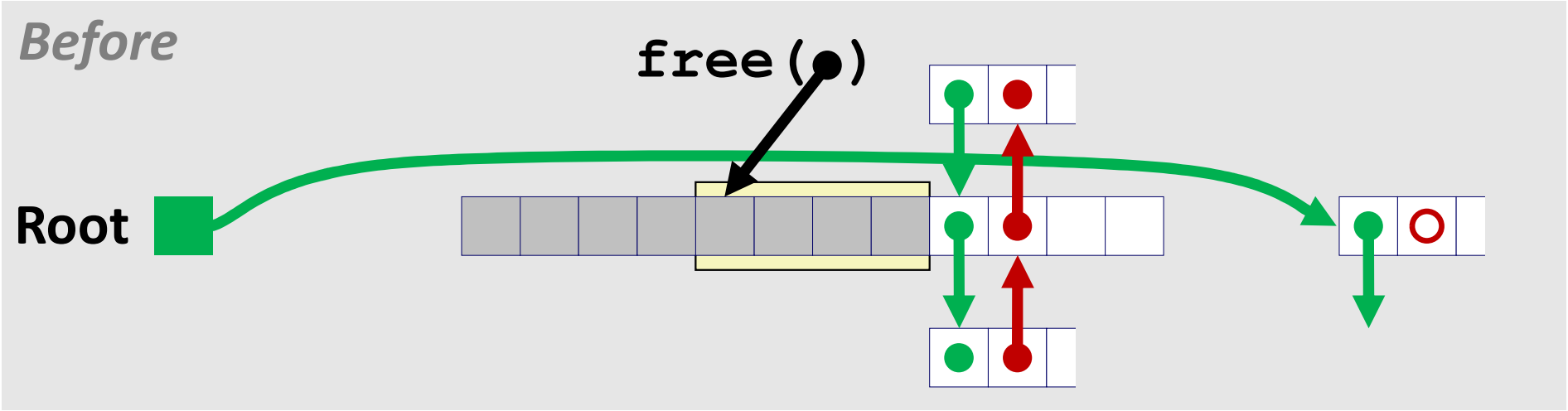
概念图



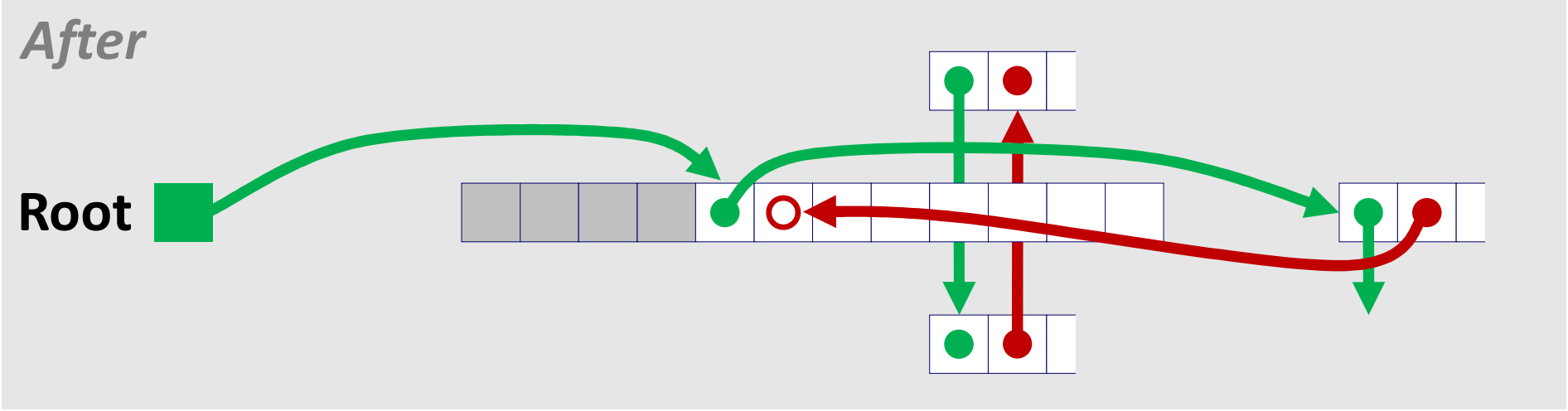
- 将释放的块插入到列表的根部



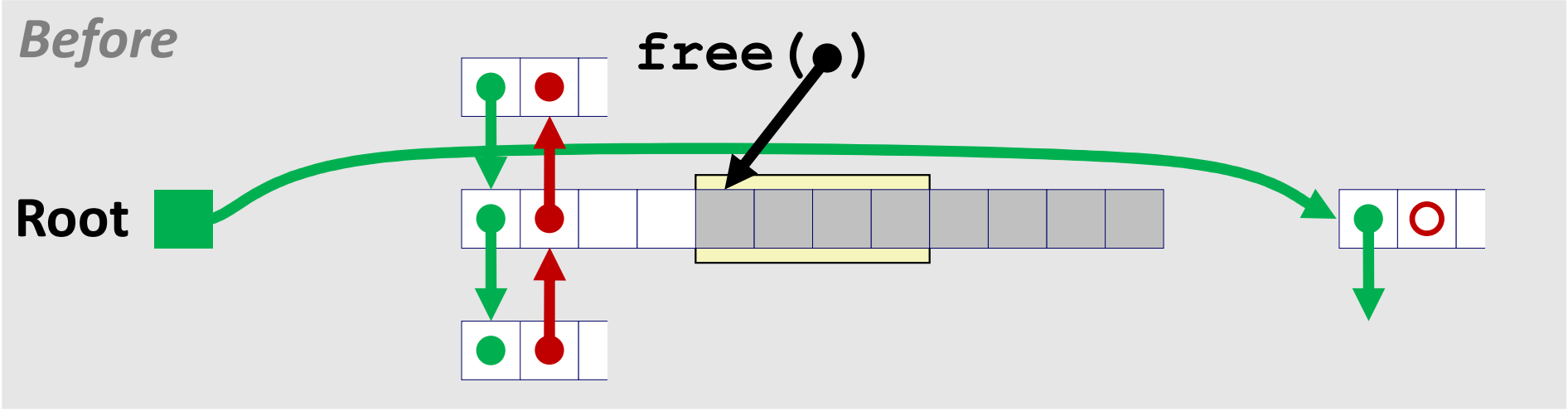
使用 LIFO 策略释放（案例 2）



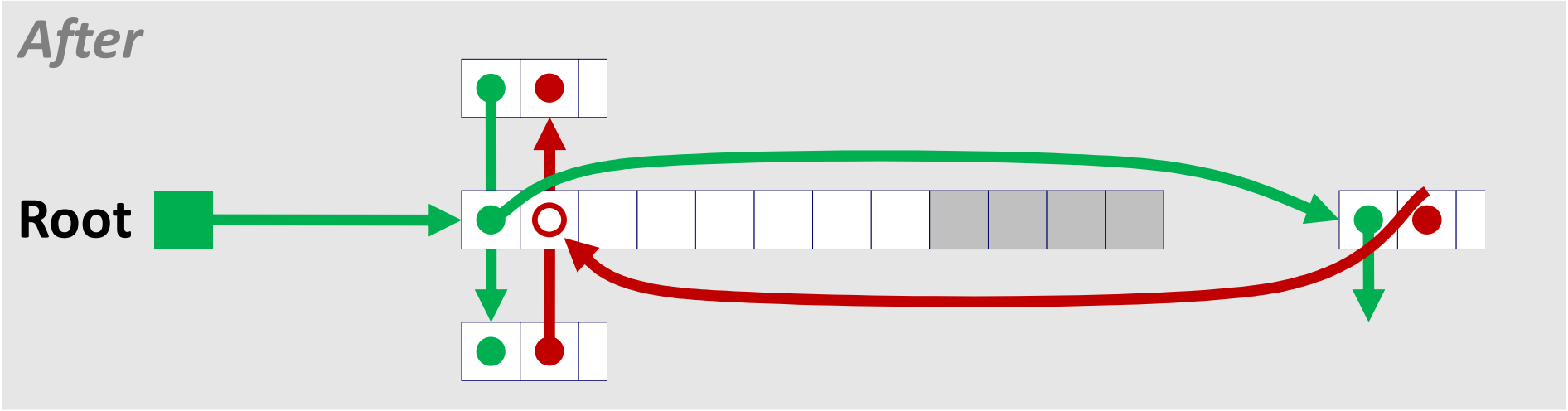
- 拼接后继块，合并两个内存块，并将新块插入列表的根部



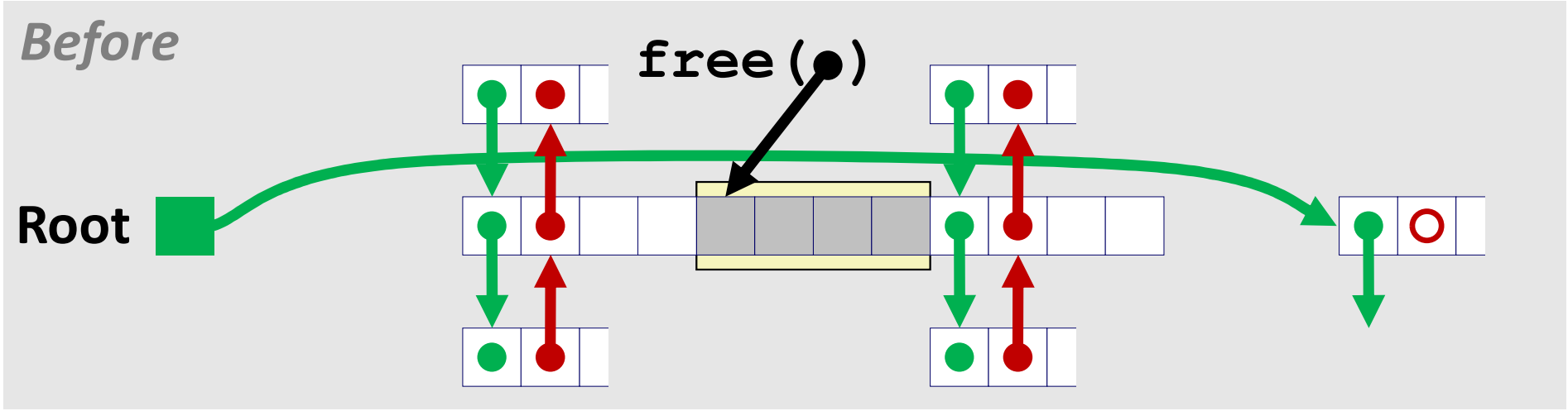
使用 LIFO 策略释放（案例 3）



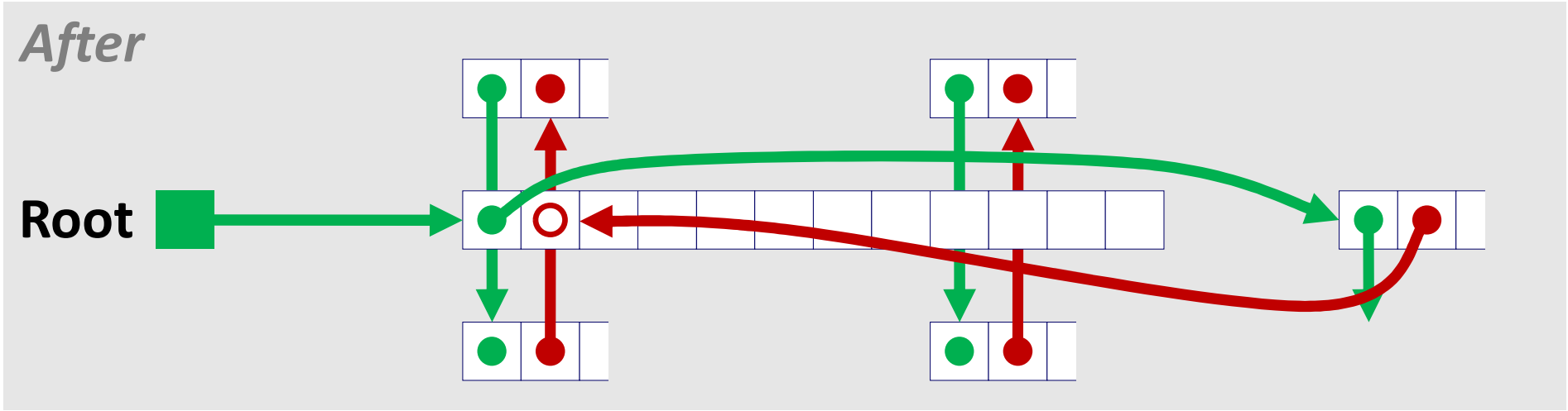
- 拼接前一个块，合并两个内存块，并将新块插入到列表的根部



使用 LIFO 策略释放（案例 4）



- 拼接前驱块和后继块，合并所有 3 个内存块，并将新块插入列表的根部



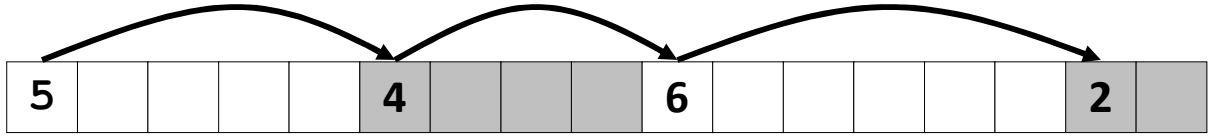
显式链表总结

- 与隐式链表的比较
  - 分配操作的时间复杂度与*空闲块*的数量成线性关系，而不是所有块。
    - 当内存大部分已被使用时，分配会快得多。
  - 分配和释放操作稍微复杂一些，因为需要把块插入或移出链表。
  - 需要额外的空间存储指针（每个块需要 2 个额外的字）。
    - 这会不会增加内部碎片？

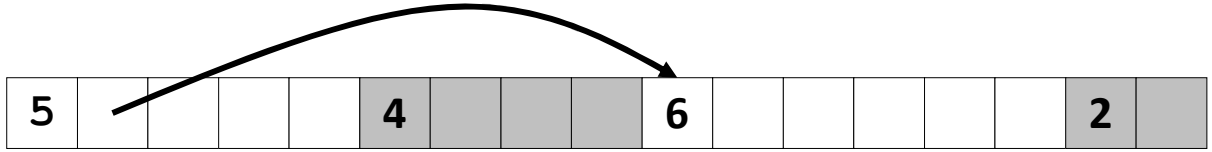
- 链表最常见的用法
  - 通常与分离空闲链表（segregated free lists）结合使用。为不同大小的块维护多个链表，或者为不同类型的对象维护多个链表。

追踪空闲块的方法

- 方法 1：隐式链表（Implicit list）使用块的长度信息，把所有块串联起来



- 方法 2：显式链表（Explicit list）仅在空闲块之间使用指针建立链表

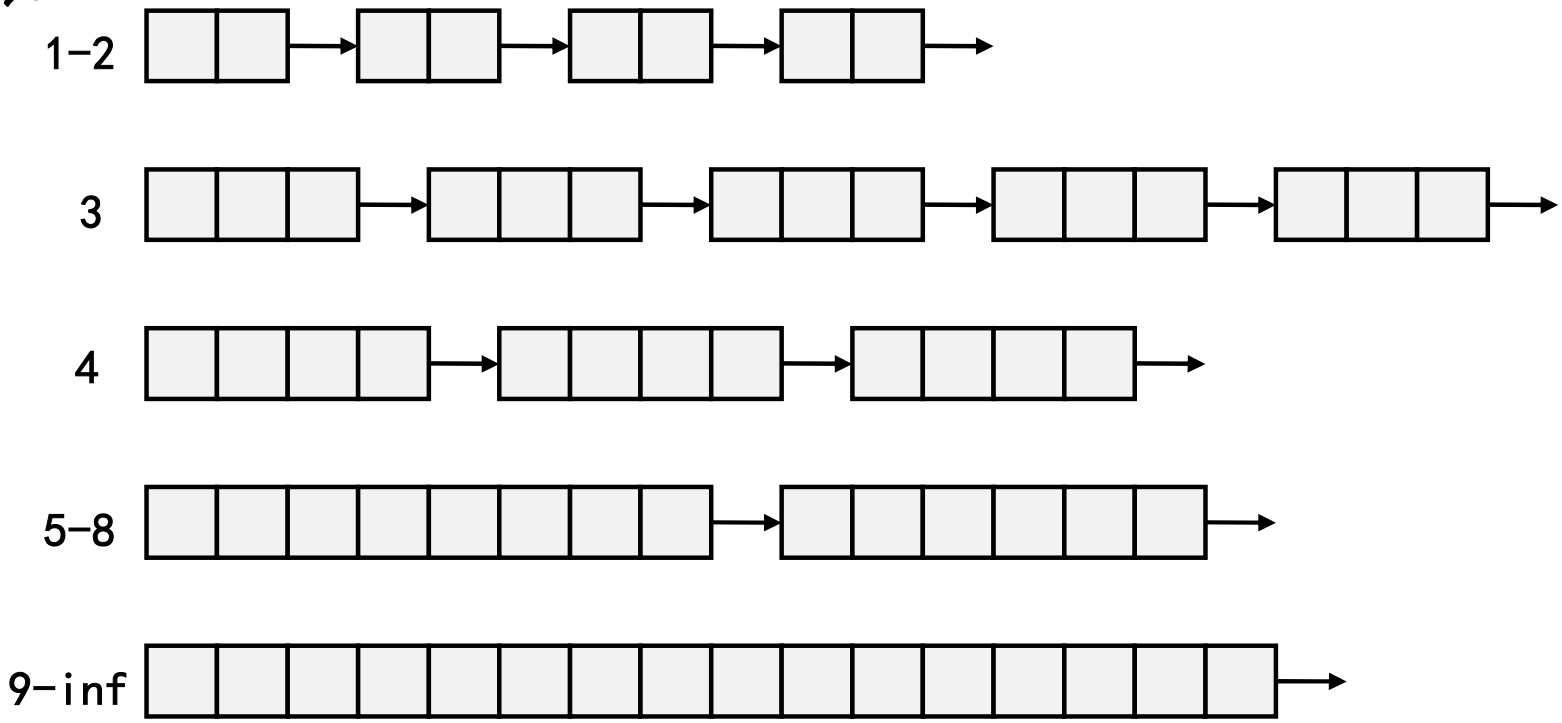


- 方法 3：分离空闲链表（Segregated free list）为不同大小类别维护不同的空闲链表

- 方法 4：按大小排序的块（Blocks sorted by size）
  - 使用平衡树（如红黑树），在每个空闲块里保存指针，并以块的长度作为关键字进行组织

分离的链表（Seglist）分配器

- 每一个 大小类别（size class） 的块都有自己独立的空闲链表



- 通常会为每一个小的大小类别单独建立一个链表
- 对于更大的块大小：通常每个 2 的幂次大小对应一个链表

分离的链表分配器（Seglist Allocator）

- 给定一个空闲链表数组，每个链表对应一个大小类

- 分配一个大小为 n 的块时：
  - 在合适的空闲链表中查找大小为 $m > n$ 的块
  - 如果找到合适的块：
    - 将块拆分，并把剩余部分放入对应的链表（可选）
  - 如果没有找到合适的块，尝试下一个更大的大小类
  - 重复以上过程，直到找到合适的块

- 如果没有找到任何块：
  - 向操作系统申请额外的堆内存（使用 sbrk()）
  - 从新获得的内存中分配 n 字节的块
  - 将剩余部分作为一个大块放入最大的大小类链表

分离的链表分配器（续）

- 释放一个块时：
  - 合并（coalesce）并将其放入合适的空闲链表中。

- Seglist 分配器的优势
  - 更高的吞吐量
    - 对于 2 的幂次方大小类，搜索时间是对数级别（log time）。
  - 更好的内存利用率
    - 在分离空闲链表上做首次适配搜索（first-fit），其效果近似于在整个堆上做最佳适配搜索（best-fit）。
    - 在极端情况下，如果给每个块分配自己对应的大小类，这就等价于最佳适配（best-fit）。

关于内存分配器的更多资料

- D. Knuth, “*The Art of Computer Programming*”, 2nd edition, Addison Wesley, 1973
  - 动态存储分配的经典参考书

- Wilson et al, “*Dynamic Storage Allocation: A Survey and Critical Review*”, Proc. 1995 Int’l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.
  - 全面性的综述

主要内容

- 显式空闲列表
- 分离空闲列表
- 垃圾回收
- 内存相关的风险和陷阱

隐式内存管理：垃圾回收

- 垃圾回收：自动回收堆内存，应用程序不需要手动释放内存

```
void foo() {
    int *p = malloc(128);
    return;  /* p块现在是垃圾 */
}
```

- 常见于许多动态语言：
  - Python, Ruby, Java, Perl, ML, Lisp, Mathematica
- 存在变体（“保守型”垃圾回收器），适用于 C 和 C++：
  - 但是，无法保证能回收所有垃圾

垃圾回收

- 内存管理器如何知道何时可以释放内存？
  - 一般情况下，我们无法知道未来会使用哪些内存，因为这取决于条件判断。
  - 但我们可以判断某些块是否无法使用，如果没有指针指向它们。
- 必须对指针做出某些假设
  - 内存管理器可以区分指针和非指针。
  - 所有指针都指向块的开始位置。
  - 不能隐藏指针
    - 例如，将其强制转换为 int，然后再转换回来。

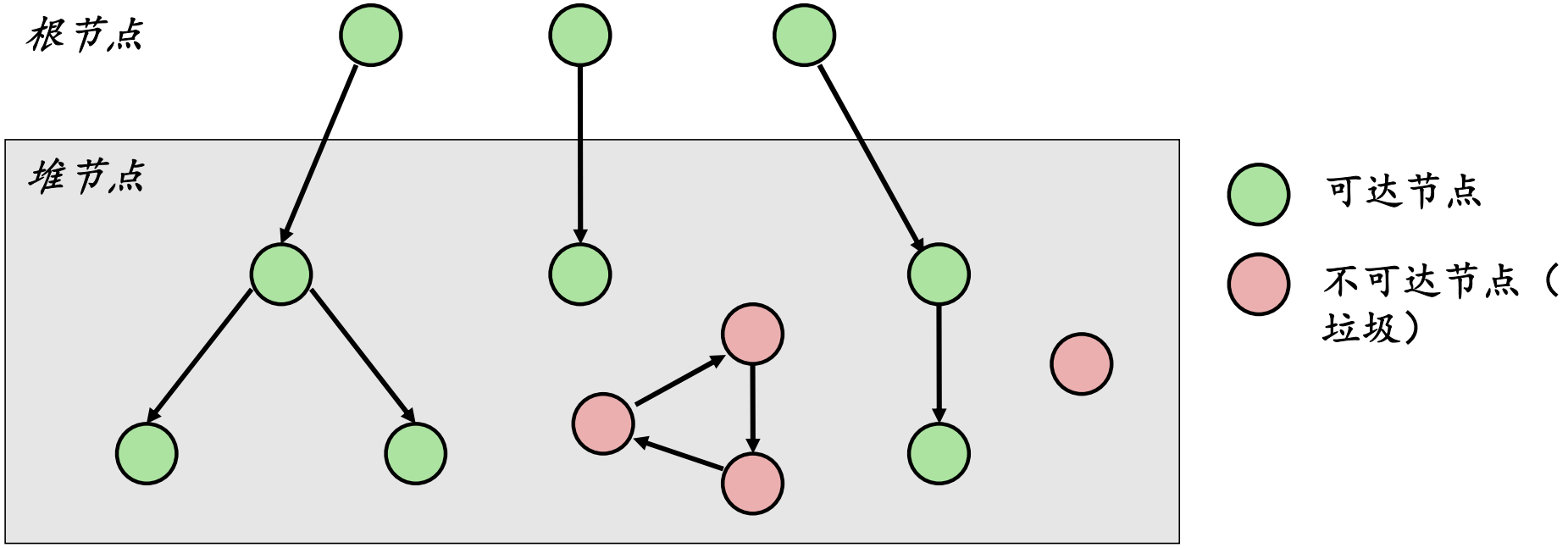
经典垃圾回收算法

- Mark-and-sweep collection (McCarthy, 1960)
  - 不会移动块（除非你还进行“压缩”操作）
- Reference counting (Collins, 1960)
  - 不会移动块（不讨论）
- Copying collection (Minsky, 1963)
  - 移动块（不讨论）
- Generational Collectors (Lieberman and Hewitt, 1983)
  - 基于生命周期的收集
    - 大多数分配很快就变成垃圾
    - 因此，集中回收最近分配的内存区域
- 更多信息:

  Jones and Lin, “*Garbage Collection: Algorithms for Automatic Dynamic Memory*”, John Wiley & Sons, 1996.

将内存视为图

- 我们将内存视为一个有向图
  - 每个块是图中的一个节点
  - 每个指针是图中的一条边
  - 堆外的位置包含指向堆的指针，被称为根节点（例如：寄存器、栈上的位置、全局变量）

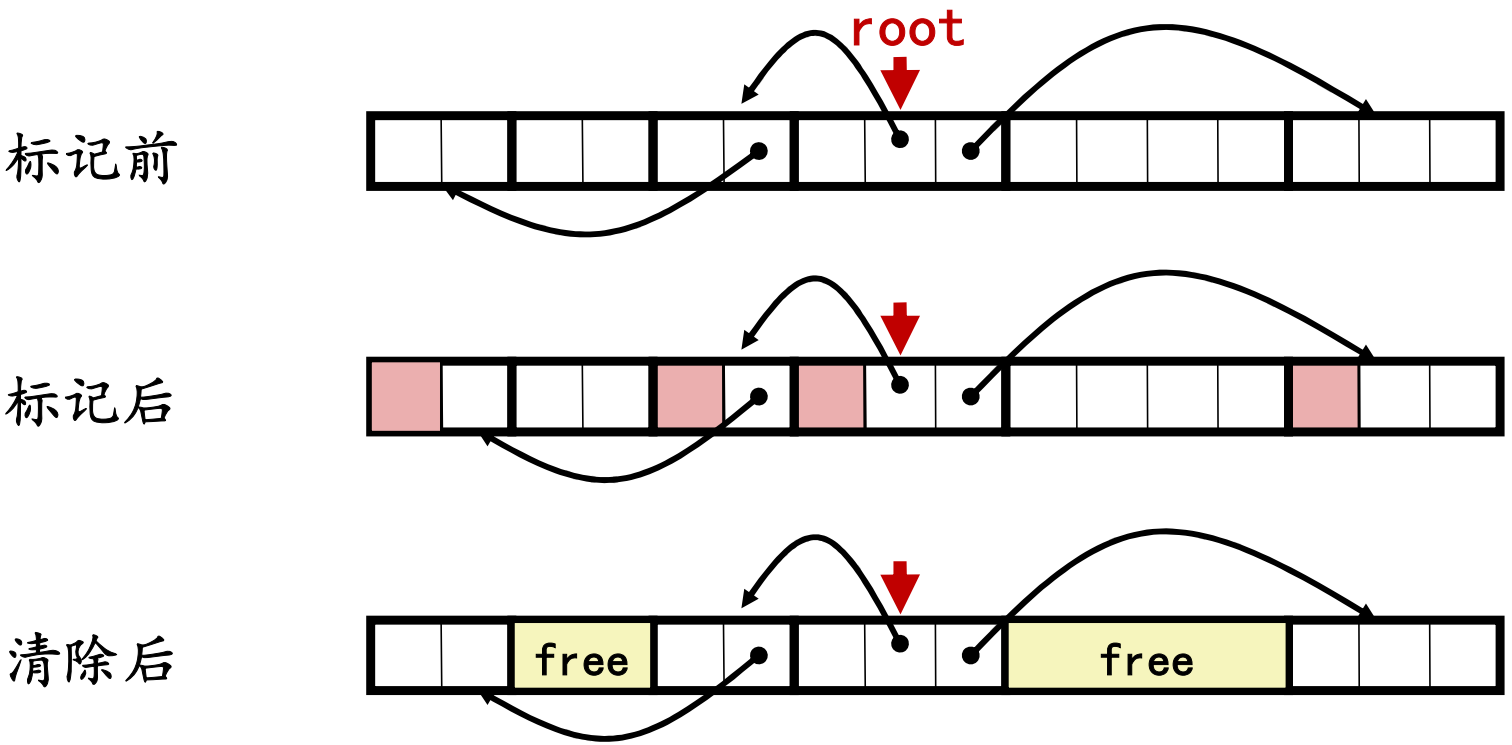


一个节点（块）是可达的，如果从任何根节点到该节点有一条路径。

不可达的节点是垃圾（不能被应用程序使用）。

标记-清除收集 Mark and Sweep Collecting

- 可以基于 malloc/free 包进行构建
  - 使用 malloc 进行分配，直到“没有空间”
- 当没有空间时：
  - 在每个块的头部使用额外的标记位
  - 标记（mark）：从根节点开始，设置每个可达块的标记位
  - 清除（sweep）：扫描所有块，释放未标记的块



注：此处箭头表示内存引用，而非空闲列表指针。

简单实现的假设

- 应用
  - new(n)：返回一个新的块指针，所有位置被清零
  - read(b, i)：将块 b 的第 i 位置读入寄存器
  - write(b, i, v)：将值 v 写入块 b 的第 i 位置
- 每个块将有一个头部字
  - 块 b 的头部位置为 b[-1]
  - 在不同的收集器中用于不同的目的
- 垃圾回收器使用的指令
  - is_ptr(p)：判断 p 是否是一个指针
  - length(b)：返回块 b 的长度，不包括头部
  - get_roots()：返回所有根节点

标记与清除（续）

使用深度优先遍历内存图进行标记

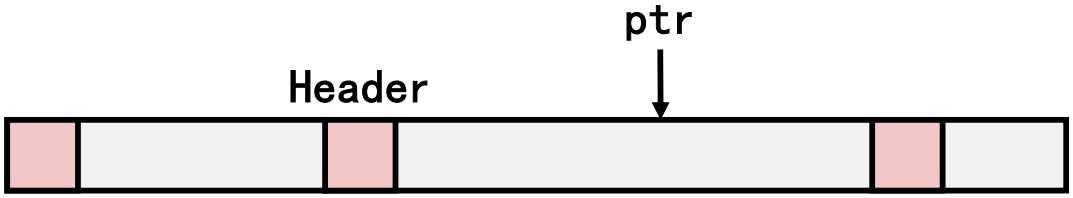
```
ptr mark(ptr p) {
  if (!is_ptr(p)) return;  // 如果不是指针，什么都不做
  if (markBitSet(p)) return;  // 检查是否已经标记
  setMarkBit(p);  // 设置标记位
  for (i = 0; i < length(p); i++)  // 对块中的所有单词进行标记
    mark(p[i]);
  return;
}
```

使用块的长度来找到下一个块进行清除

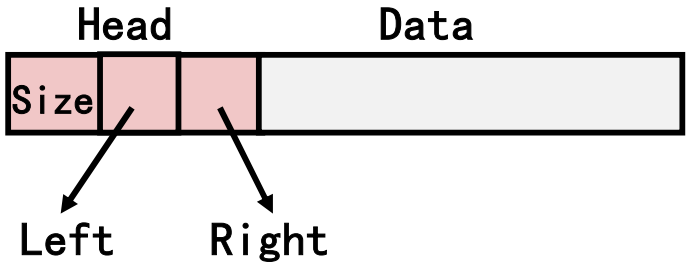
```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
}
```

C语言中的保守标记与清除（Mark & Sweep）

- C程序的“保守垃圾回收器”
  - is_ptr()：通过检查是否指向一个已分配的内存块来判断一个词是否是指针
  - 但是，在C语言中，指针可以指向块的中间位置



- 如何找到块的开始位置？
  - 可以使用平衡二叉树来跟踪所有已分配的块（关键是块的起始位置）
  - 平衡树指针可以存储在头部（使用两个额外的字）



左边：较小的地址
右边：较大的地址

主要内容

- 显式空闲列表
- 分离空闲列表
- 垃圾回收
- 内存相关的风险和陷阱

与内存相关的危险与陷阱

- 解引用坏指针
- 读取未初始化的内存
- 覆写内存
- 引用不存在的变量
- 多次释放块
- 引用已释放的块
- 未能释放块

解引用坏指针

- 经典的 scanf 错误

```
int val;

...

scanf(“%d”, val);
```

```
int val;

...

scanf(“%d”, &val);
```

读取未初始化的内存

- 假设堆数据被初始化为零

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc(N*sizeof(int));
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```

读取未初始化的内存

- 假设堆数据被初始化为零

```
/* return y = Ax */
int *matvec(int **A, int *x) {
    int *y = malloc(N*sizeof(int));
    int i, j;

    for (i=0; i<N; i++)
        y[i] = 0;  // 初始化 y[i] 为零
        for (j=0; j<N; j++)
            y[i] += A[i][j]*x[j];
    return y;
}
```

覆写内存

- 覆写内存
- 没有检查最大字符串大小

```
char s[8];
int i;
gets(s);  /* 从 stdin 读取 "123456789" */
```

- 经典缓冲区溢出攻击的基础

覆写内存

- 覆写内存
- 没有检查最大字符串大小

```
char s[8];
int i;
gets(s);  /* 从 stdin 读取 "123456789" */
```

- 经典缓冲区溢出攻击的基础

- fgets(s, sizeof(s), stdin); // 限制最大读取长度，防止溢出

引用不存在的变量

- 忘记局部变量在函数返回时会消失

```
int *foo () {
   int val;

   return &val;
}
```

多次释放块

```
x = malloc(N*sizeof(int));
        <manipulate x>
free(x);

y = malloc(M*sizeof(int));
        <manipulate y>
free(x);
```

引用已释放的块

```
x = malloc(N*sizeof(int));
  <manipulate x>
free(x);
   ...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

未释放块（内存泄漏）

- 缓慢的长期杀手！

```
foo() {
    int *x = malloc(N*sizeof(int));
    ...
    return;
}
```

处理内存错误

- 调试器：gdb
  - 适合用于查找坏指针解引用错误
  - 难以检测其他内存错误
- 数据结构一致性检查器
  - 静默运行，仅在发生错误时打印信息
  - 用作错误定位的探测工具
- 二进制翻译器：valgrind
  - 强大的调试和分析技术
  - 重写可执行目标文件的文本部分
  - 运行时检查每个单独的引用
    - 坏指针、覆盖、超出分配块的引用
- glibc malloc 包含检查代码
  - setenv MALLOC_CHECK_ 3