第9章 虚拟内存：
动态内存分配 ——高级主题

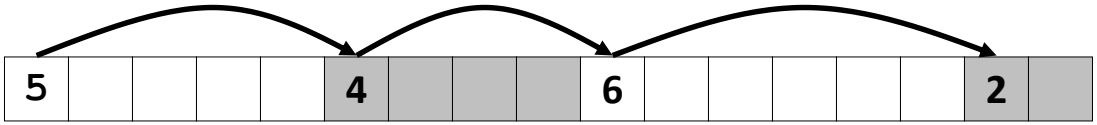
教 师： 郑贵滨
计算机科学与技术学院
哈尔滨工业大学

主要内容

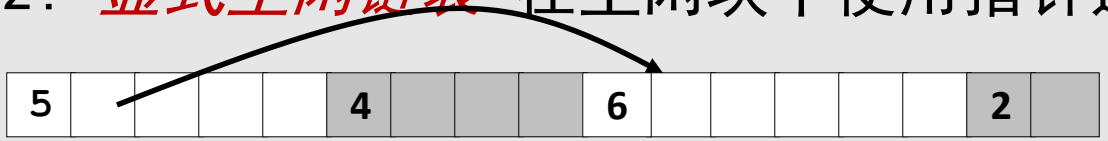
- 显式空闲链表(Explicit free lists）
- 分离的空闲链表(Segregated free lists)
- 垃圾收集(Garbage collection)
- 内存相关的风险和陷阱(Memory-related perils and pitfalls)

跟踪空闲块

- 方法 1: *隐式空闲链表* 通过头部中的大小字段隐含地连接空闲块

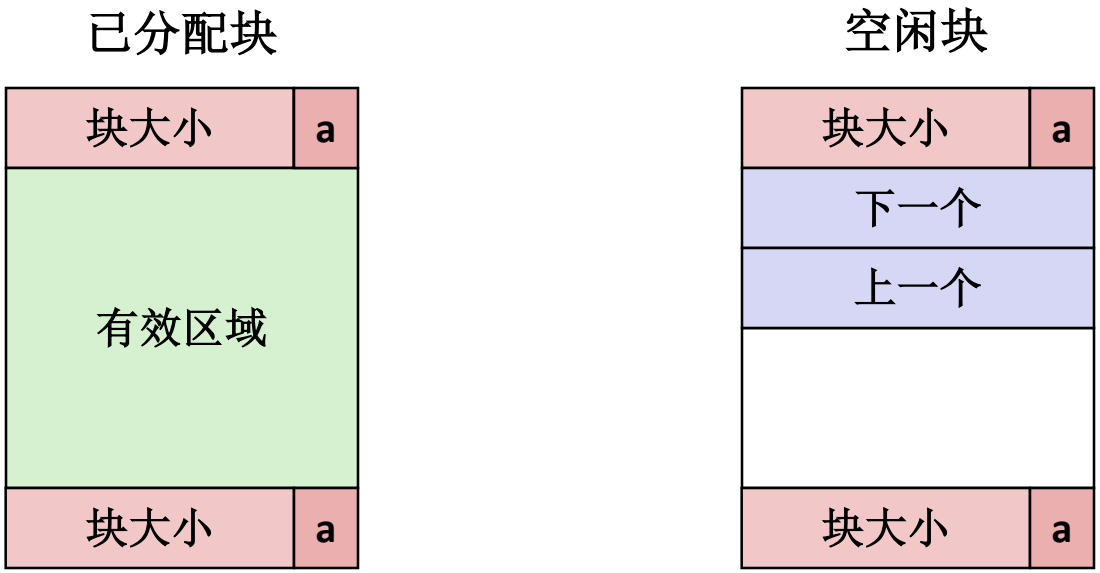


- 方法 2: *显式空闲链表* 在空闲块中使用指针连接空闲块



- 方法 3: *分离的空闲链表*
  - 每个大小类的空闲链表包含大小相等的块
- 方法 4: *按照尺寸排序的块*
  可以使用平衡树（例如红黑树），在每个空闲块中有指针，尺寸作为键。

显式空闲链表

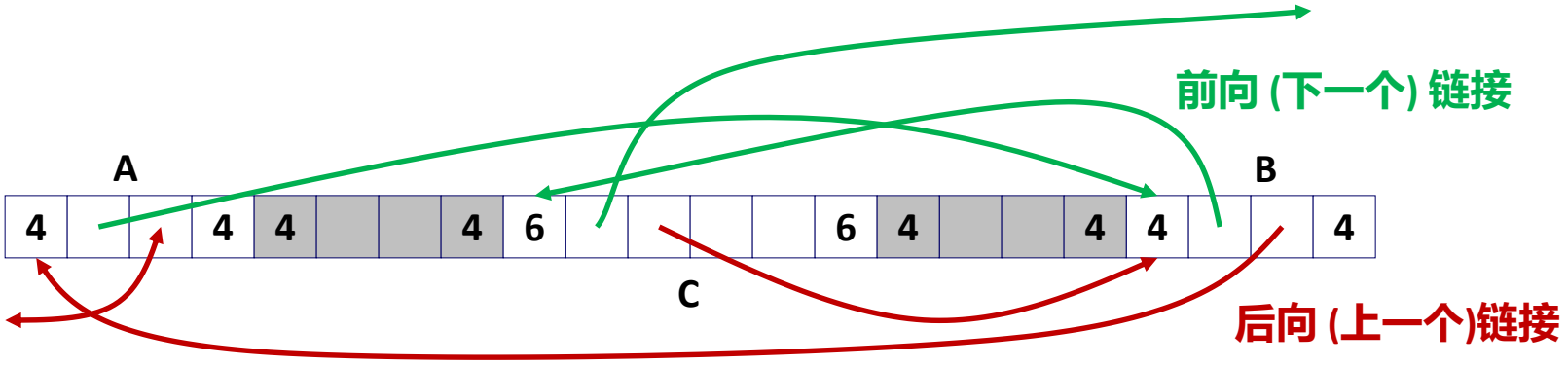


- 维护*空闲块*链表，而不是*所有块*
  - “下一个”空闲块可以在任何地方
    - 因此需要存储前/后指针，而不仅仅是大小（size）
  - 还需要边界标记，用于块合并
  - 幸运的是，只需跟踪空闲块，因此可以使用有效载荷区域。

显式空闲链表

- 逻辑层面

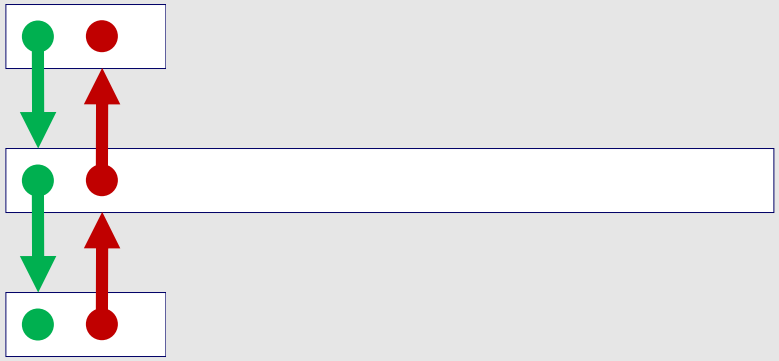
- 物理层面: 块的顺序是任意的



显式空闲链表——分配

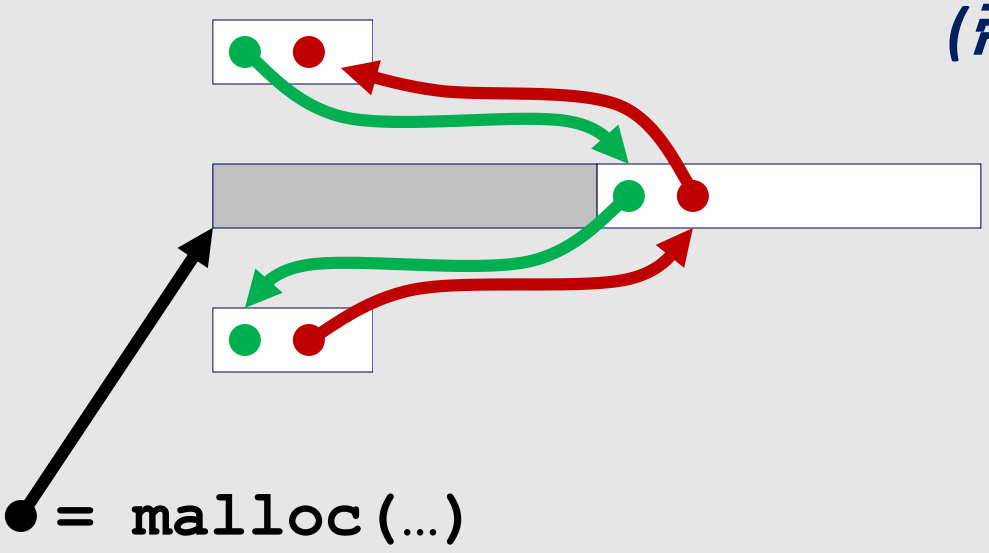
分配示意图

分配前



分配后

(带分割的分配)

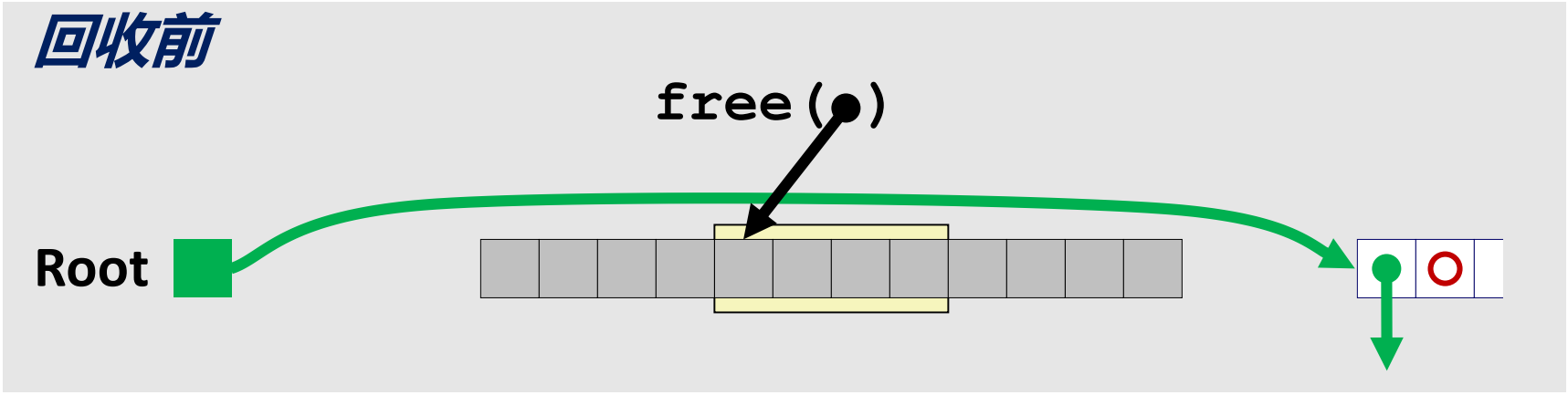


显式空闲链表——释放

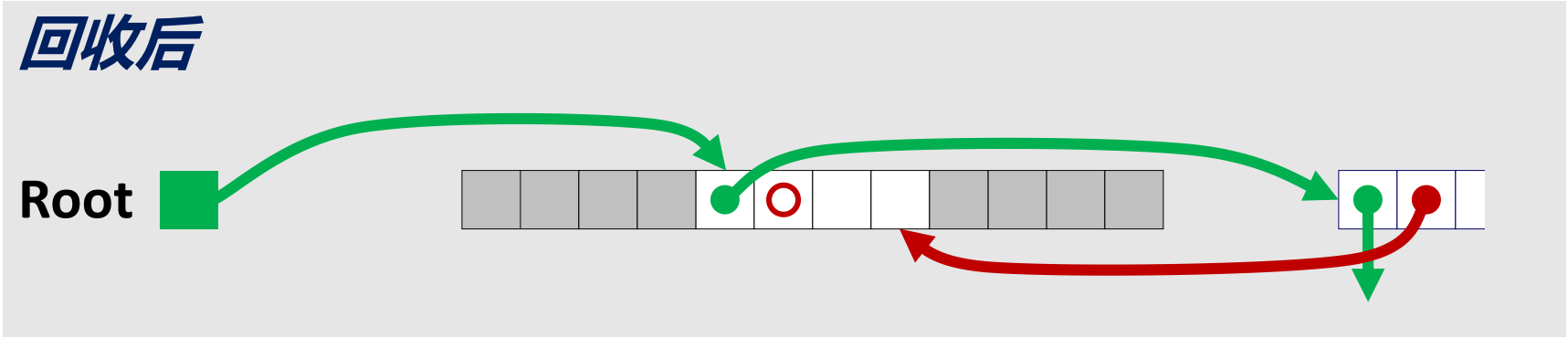
- *插入原则:*
  一个新释放的块放在空闲链表的什么位置？
- LIFO (last-in-first-out) 策略：后进先出法
  - 将新释放的块放置在链表的开始处
  - *优点:* 简单，常数时间
  - *缺点:* 研究表明碎片比地址顺序法更糟糕
- 地址顺序法(Address-ordered policy)
  - 按照地址顺序维护链表:
    addr(前一个块) < addr(当前回收块) < addr(下一个块)
  - *优点:* 研究表明碎片要少于LIFO (后进先出法)
  - *缺点:* 需要搜索

LIFO (后进先出) 回收策略 (情况1)

概念图

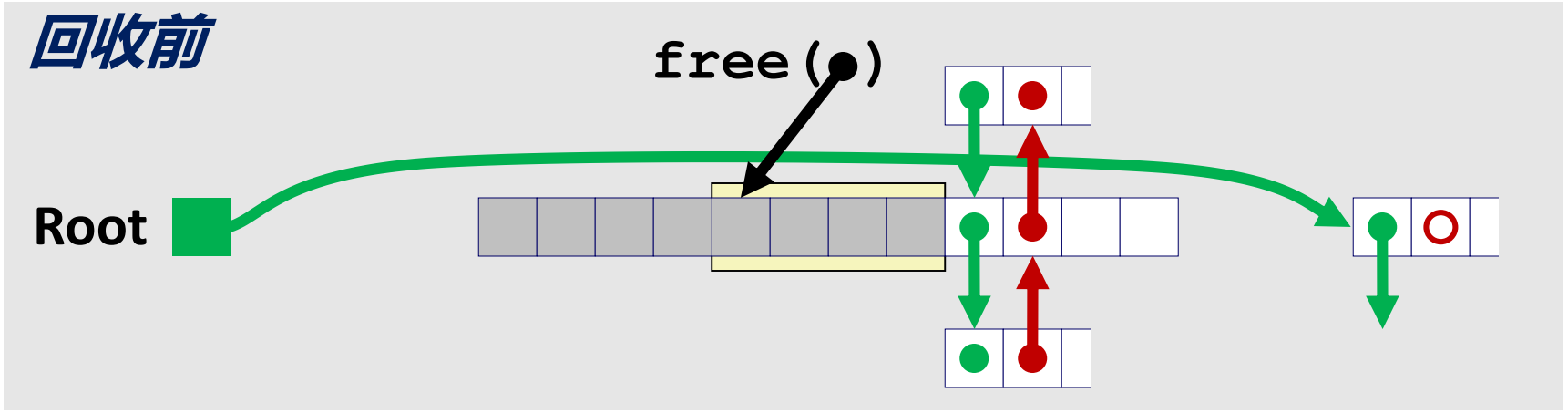


- 将新释放的块放置在链表的开始处

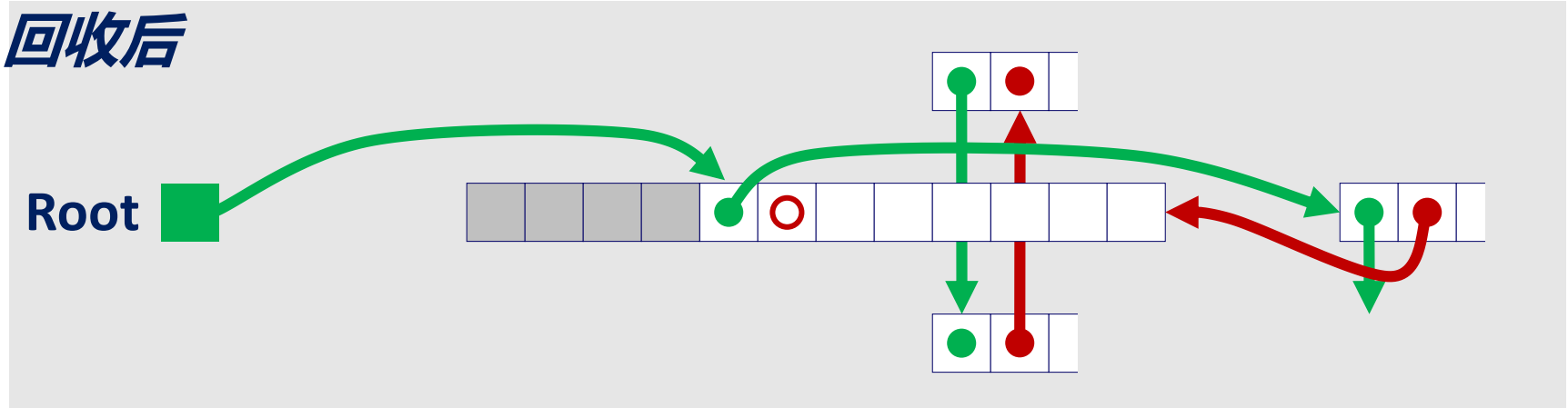


LIFO (后进先出) 回收策略 (情况2)

概念图

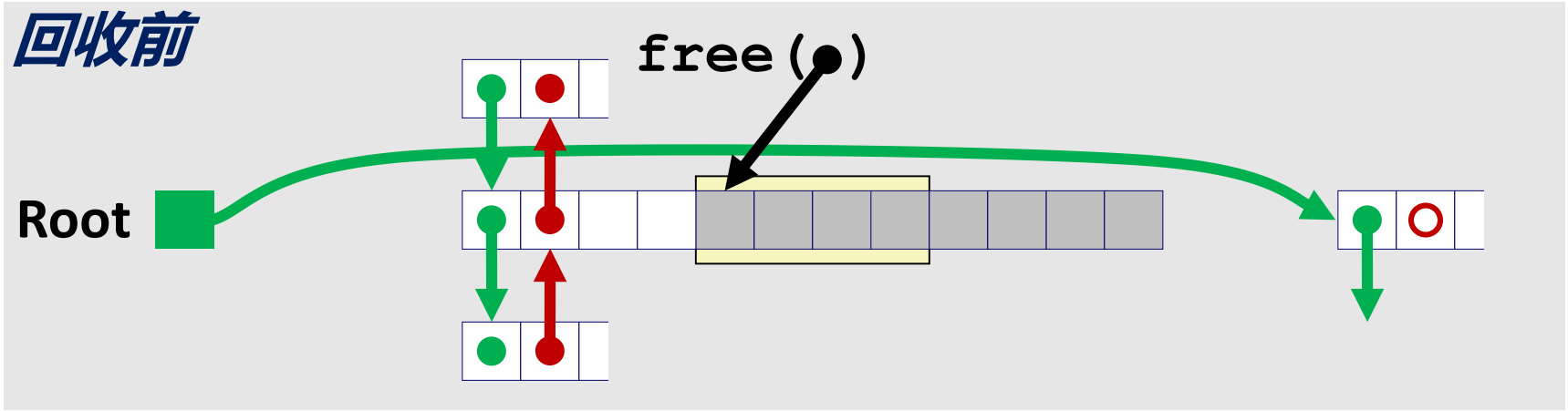


- 与后继块合并成一个新块（2块合并），并插入到链接表的开始处

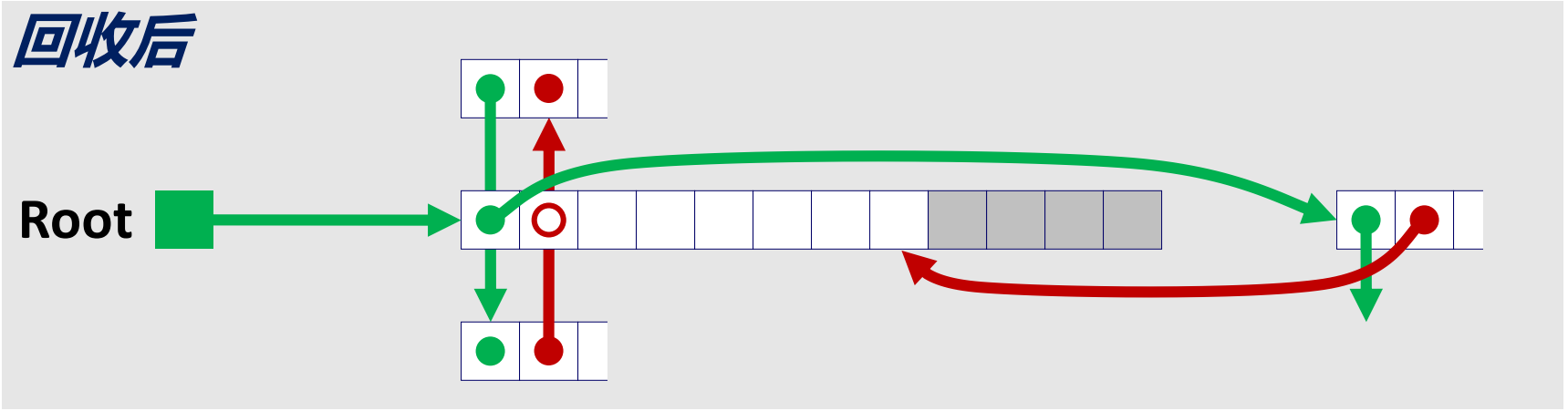


LIFO (后进先出) 回收策略 (情况3)

概念图

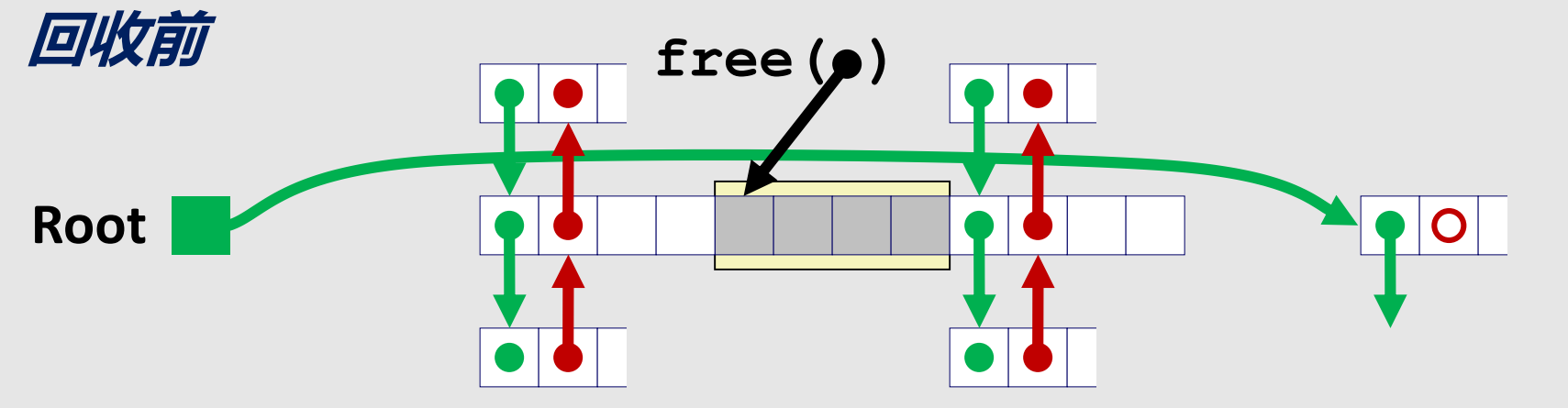


- 与前序空闲块合并成一个新空闲块（2块合并），并插入到空闲链表的开始处

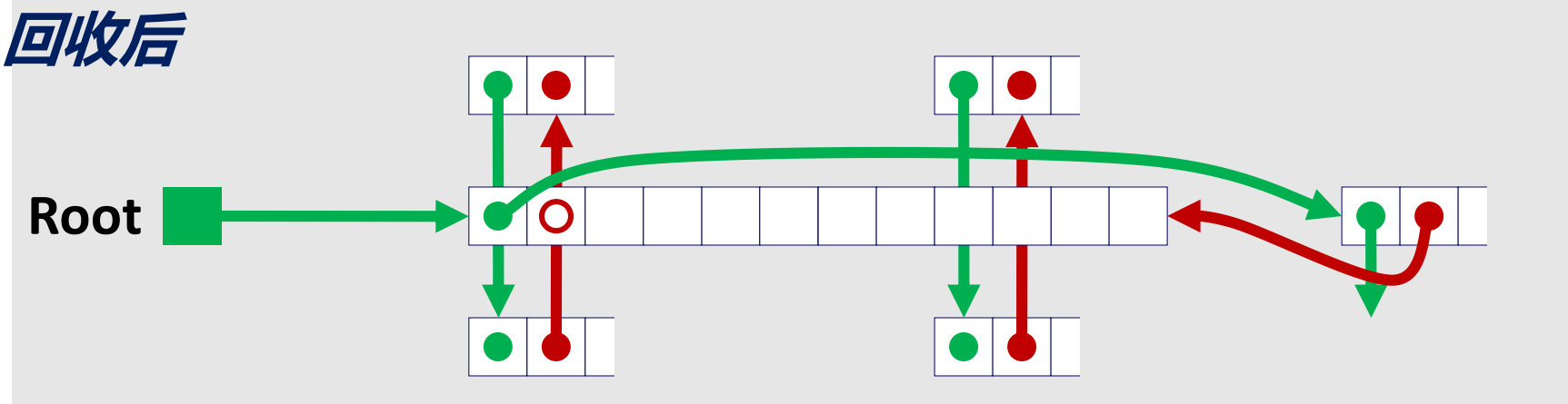


LIFO (后进先出) 回收策略 (情况4)

概念图



- 与前块和后继块合并成一个新空闲块（3块合并），插入到链表的开始

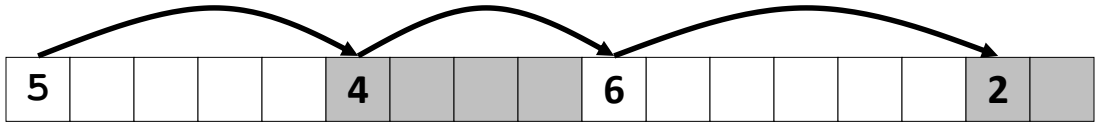


显式空闲链表小结

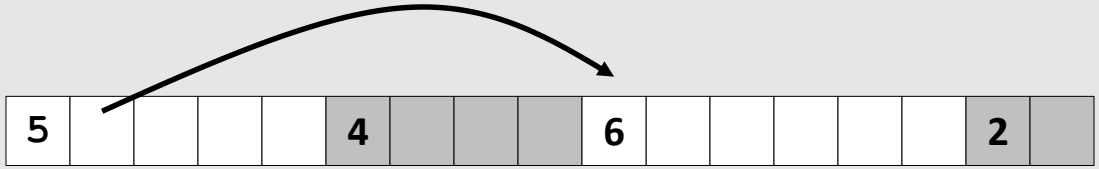
- 与隐式链表相比较:
  - 分配时间：从块总数的线性时间减少到空闲块数量的线性时间
    - 当大量内存被占用时*快得多*
    - 需要在链表中拼接块，释放和分配稍显复杂一些
- *额外的链接空间开销:*每个块需要2个额外的字
  - 这样会不会增加内部碎片？
    - 更大的最小块大小、潜在地提高了内部碎片的程度
- 最常用的链表：分离的空闲链表
  - 维护多个空闲链表，每个链表中的空闲块尺寸大致相等，或者是不同对象类型。

跟踪空闲块

- 方法 1: *隐式空闲链表* 通过头部中的大小字段隐含地连接空闲块



- 方法 2: *显式空闲链表* 在空闲块中使用指针连接空闲块



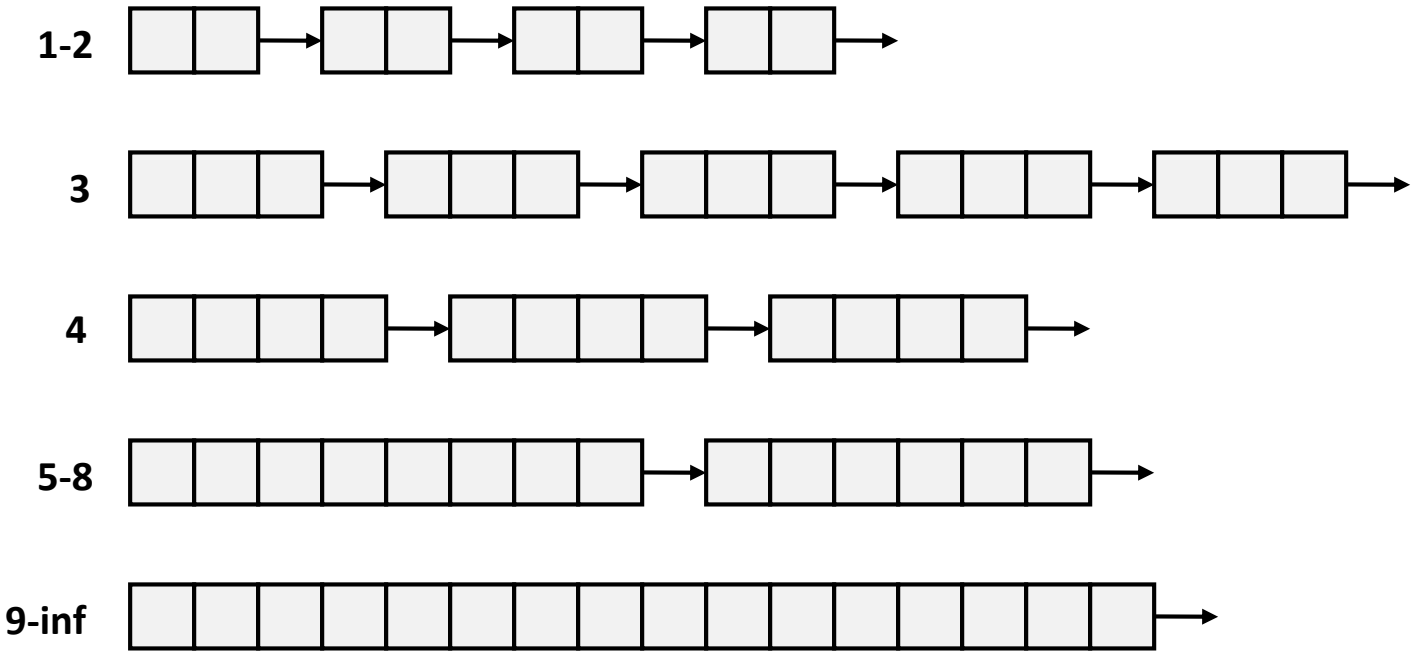
- 方法 3: *分离的空闲链表*
  - 每个大小类的空闲链表包含大小相等的块
- 方法 4: *按照尺寸排序的块*
  - 可以使用平衡树（例如红黑树），在每个空闲块中有指针，尺寸作为键。

主要内容

- 显示空闲链表(Explicit free lists)
- 分离的空闲链表(Segregated free lists)
- 垃圾收集（Garbage collection）
- 内存相关的风险和陷阱(Memory-related perils and pitfalls)

分离空闲链表分配器

- 每个*大小类(size class)*中的块构成一个空闲链表



- 通常每个小的尺寸/大小/size，都是一个单独的类
- 对于大的尺寸/大小/size：按照2的幂分类

分离适配

- 分配器维护空闲链表数组，每个空闲链表和一个大小类关联，链表是显示或隐式的。
- 当分配器需要一个大小为n的块时:
  - 搜索相应的空闲链表，其大小要满足m > n
  - 如果找到了合适的块:
    - 拆分块，并将剩余部分插入到适当的可选列表中
  - 如果找不到合适的块, 就搜索下一个更大的大小类的空闲链表
  - 直到找到为止。
- 如果空闲链表中没有合适的块:
  - 向操作系统请求额外的堆内存 (使用sbrk())
  - 从这个新的堆内存中分配出 n 字节
  - 将剩余部分放置在适当的大小类中

分离适配

- 释放块
  - 合并，并将结果放置到相应的空闲链表中
- 分离适配的优势
  - 更高的吞吐量
    - ***log time for power-of-two size classes 按照2的幂得到各类使用log时间***
  - 更高的内存使用率
    - 对分离空闲链表的简单的首次适配搜索，其内存利用率近似于对整个堆的最佳适配搜索的内存利用率。
    - 极端情况：如果每个块都属于它本身尺寸的大小类，那么就相当于最佳适应算法。

关于分配器的更多信息

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

- Wilson et al, "*Dynamic Storage Allocation: A Survey and Critical Review*", Proc. 1995 Int'l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.
  - 全面的综述
  - 可以从 CS:APP student site (csapp.cs.cmu.edu) 获取

主要内容

- 显示空闲链表(Explicit free lists)
- 分离的空闲链表(Segregated free lists)
- 垃圾收集（Garbage collection）
- 内存相关的风险和陷阱(Memory-related perils and pitfalls)

隐式内存管理——垃圾收集

- *垃圾收集:* 自动回收堆存储的过程—应用从不显式释放

```
void foo() {
    int *p = malloc(128);
    return; /* p block is now garbage */
}
```

- 常见于多种动态语言中:
  - Python, Ruby, Java, Perl, ML, Lisp, Mathematica
- C 和 C++ 有变种的垃圾收集器——保守的垃圾收集器
  - 因为保守，不能确保收集所有的垃圾

垃圾收集

- 内存管理器如何知道何时可以释放内存？
  - 一般我们不知道将来会用到什么，因为这取决于具体条件
  - 但可以确定：
    如果没有指针指向一个块，那这个块就不能被使用

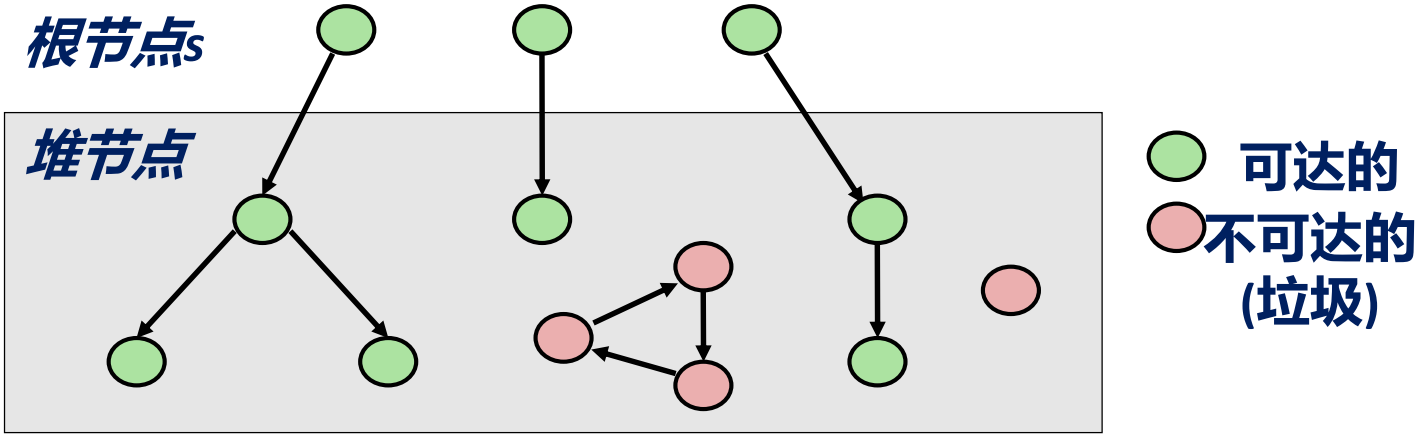
- 必须做些关于指针的假设
  - 内存管理器可以区分指针和非指针
  - 所有指针都指向一个块的起始地址
  - 无法隐藏指针
    (例如：将指针强制转换成int，然后再转换成指针)

经典的垃圾收集算法

- 标记&清除(Mark&Sweep)垃圾收集器(McCarthy, 1960)
  - 不移动块 (除非要 “压缩”)
- 引用计数Reference counting (Collins, 1960)
  - 不移动块 (不讨论)
- 复制收集Copying collection (Minsky, 1963)
  - 移动块 (不讨论)
- Generational Collectors (Lieberman and Hewitt, 1983)
  - 基于生命期的收集
    - 大部分分配很快就会变成垃圾
    - 因此回收工作的重点应该是刚刚分配的内存区域
- 获得更多信息:
  Jones and Lin, “*Garbage Collection: Algorithms for Automatic Dynamic Memory*”, John Wiley & Sons, 1996.

将内存视为有向图

- 将内存看作一张有向图
  - 每个块是图中的一个节点
  - 每个指针是图中的一个边
  - 根节点的位置一定不在某些堆中，这些堆中包含指向堆的指针（这些位置可以是寄存器、栈里的变量，或者是虚拟内存中读写数据区域的全局变量）

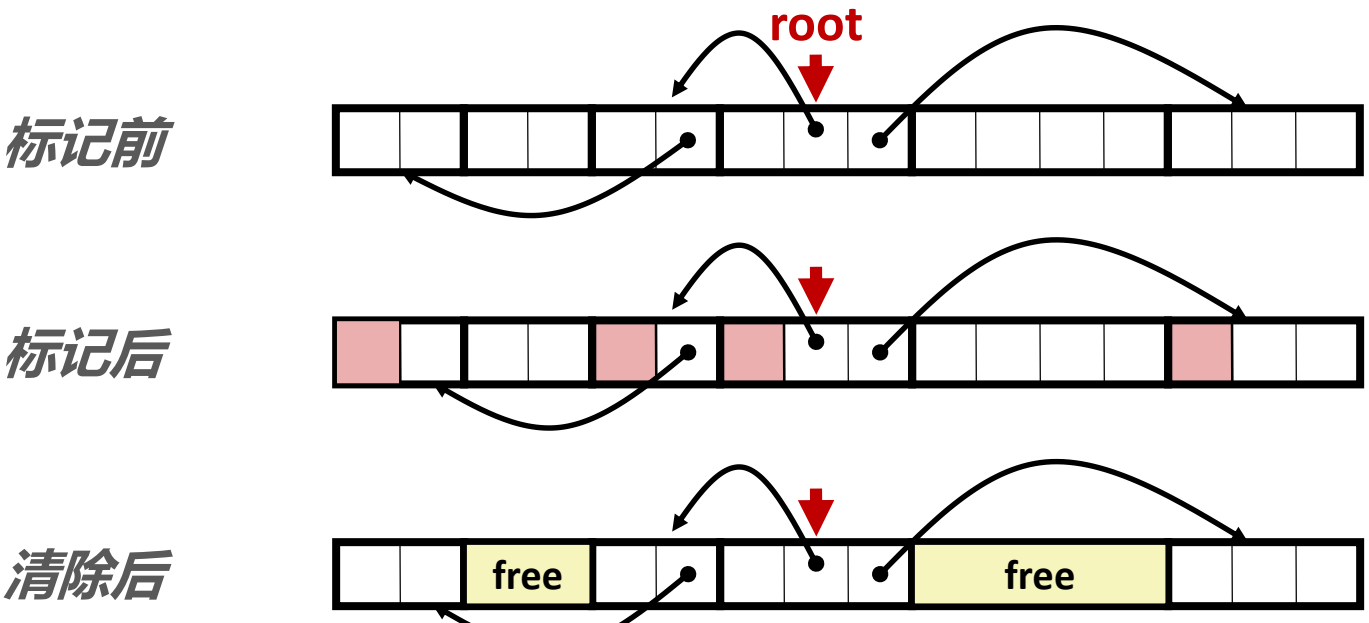


*可达*节点(块)：存在一条从任意根节点出发并到达该节点的有向路径

不可达节点是*垃圾*（不能被应用程序再次使用）

标记&清除垃圾收集器

- 可以建立在已存在的malloc包的基础上
  - 使用malloc分配直到“用完了空间”
- 当“用完了空间”：
  - 使用块头部中的 ***mark bit标记位***
  - ***标记:*** 从根节点开始标记所有的可达块
  - ***清除:*** 扫描所有块并释放没有被标记的块



注意：箭头表示内存引用，而不是空闲链表指针

标记位设置

简单实现的假设

- 应用
  - **new(n):** 返回指向所有位置已被清除的新块的指针
  - **read(b,i):** 读取 **b** 块位置 **i** 的内容到寄存器
  - **write(b,i,v):** 将内容 **v** 写入到 **b** 块位置 I
- 每个块都会有一个包含一个字的头部
  - 对于块**b**，标记为 **b[-1]**
  - 用在不同的收集器中，可以起到不同的作用
- 垃圾收集器使用函数的说明
  - **is_ptr(p):** 判断p是不是指针
  - **length(b):** 返回块b的以字为单位的长度（不包括头部）
  - **get_roots():** 返回所有根节点

标记&清除...

Mark（标记） 使用内存图的深度优先遍历

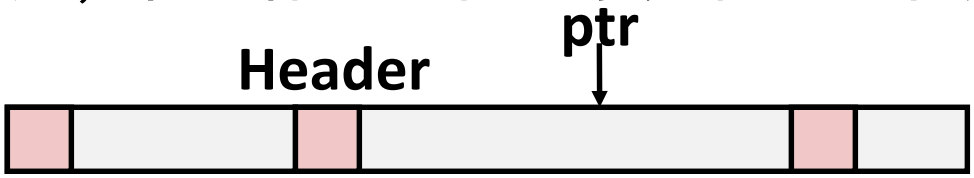
```
ptr mark(ptr p) {
    if (!is_ptr(p)) return;          // 不是指针则什么都不做
    if (markBitSet(p)) return;       // 检查是否已标记
    setMarkBit(p);                   // 设置标记位
    for (i=0; i < length(p); i++)    // 调用mark标记块中
      mark(p[i]);                    // 的每个字
    return;
}
```

Sweep （清除） 使用长度查找下一个块

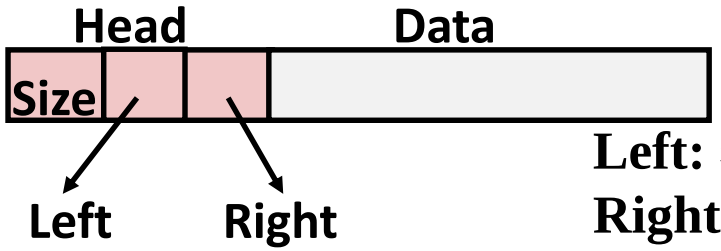
```
ptr sweep(ptr p, ptr end) {
    while (p < end) {
        if markBitSet(p)
            clearMarkBit();
        else if (allocateBitSet(p))
            free(p);
        p += length(p);
}
```

C语言：保守的Mark & Sweep

- C程序的“保守的”垃圾收集器
  - **is_ptr()** :通过检查某个字是否指向已分配的内存块来确定该字是否为指针
  - 但是，在C语言中指针可以指向一个块的中间位置



- 如何找到块的起始位置?
  - 可以使用平衡二叉树跟踪所有分配的块
  - 平衡树指针可以存储在每个已分配块的头部（使用两个额外的字left和right）



主要内容

- 显示空闲链表(Explicit free lists)
- 分离的空闲链表(Segregated free lists)
- 垃圾收集（Garbage collection）
- 内存相关的风险和陷阱(Memory-related perils and pitfalls)

内存相关的风险和陷阱

- 解引用（间接引用）坏指针
- 读未初始化的内存
- 覆盖内存
- 引用不存在的变量
- 多次释放内存
- 引用空闲堆块中的数据
- 释放内存失败

C 运算符

- ->, (), [] 有高优先级，* 和 & 次之
- 一元 +，-, * 比二元形式有更高的优先级

运算符	结合性
() [] -> .	从左到右
! ~ ++ -- + - * & (type) sizeof	从右到左
* / %	从左到右
+ -	从左到右
<< >>	从左到右
< <= > >=	从左到右
== !=	从左到右
&	从左到右
^	从左到右
|	从左到右
&&	从左到右
||	从左到右
?:	从右到左
= += -= *= /= %= &= ^= != <<= >>=	从右到左
,	从左到右

C 指针的使用: 来做个自测吧!

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*f())[13])()`	f is a function returning ptr to an array[13] of pointers to functions returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints

Source: K&R Sec 5.12

间接引用坏指针

- 经典的scanf错误

```
int val;

...

scanf("%d", val);
```

读未初始化的内存

- 常见的错误是假设堆内存被初始化为零

```
/* 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;
}
```

覆盖内存

- 分配（可能）错误大小的对象

```
int **p;

p = malloc(N*sizeof(int));

for (i=0; i<N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

假设指向对象的指针和它们所指向的对象是相同大小的

覆盖内存

- Off-by-one error 错位错误

```
int **p;

p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

覆盖内存

- 不检查输入串的大小

```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

- 经典缓冲区溢出攻击的基础

覆盖内存

- 误解指针运算

```
int *search(int *p, int val) {

   while (*p && *p != val)
      p += sizeof(int);

   return p;
}
```

覆盖内存

- 引用指针而不是它所指向的对象

```
int *BinheapDelete(int **binheap, int *size) {
   int *packet;
   packet = binheap[0];
   binheap[0] = binheap[*size - 1];
   *size--;
   Heapify(binheap, *size, 0);
   return(packet);
}
```

引用不存在的变量

- 忘记当函数返回时局部变量将消失

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

    return &val;
}
```

多次释放

- 很讨厌!

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

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

引用空闲堆块中的数据

- 太邪恶!

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

释放失败（内存泄漏）

- 慢性、长期的杀手!

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

释放失败（内存泄漏）

- 释放部分数据

```
struct list {
    int val;
    struct list *next;
};

foo() {
    struct list *head = malloc(sizeof(struct list));
    head->val = 0;
    head->next = NULL;
    <create and manipulate the rest of the list>
     ...
    free(head);
    return;
}
```

处理内存bug

- 调试器: gdb
  - 有利于发现间接引用坏指针
  - 很难检测其他内存bug
- 数据一致性检查
  - 安静运行，仅在错误时打印消息
  - 用作探针，以发现并消除错误
- 二进制转换器valgrind
  - 强大的调试和分析技术
  - 重写可执行目标文件的文本段
  - 运行时检查每个引用
    - 坏指针、覆盖写、引用已分配块之外的内容

处理内存bug

- GNU标准库(glibc)的内存检查
  - 通过环境变量MALLOC_CHECK_，设定内置的调试特性，对动态内存进行调试。
  - 在默认情况下是不设定的,在老的版本默认这个值为0,新的版本默认值为2,但有一个矛盾,如果设定为空,它将会打印出长长的跟踪信息,这比设为2更详细。
  - MALLOC_CHECK_有三种设定:
    - MALLOC_CHECK_=0 ——关闭所有检查。
    - MALLOC_CHECK_=1 ——当有错误被探测到时,在标准错误输出(stderr)上打印错误信息。
    - MALLOC_CHECK_=2 ——当有错误被探测到时,不显示错误信息,直接进行中断。