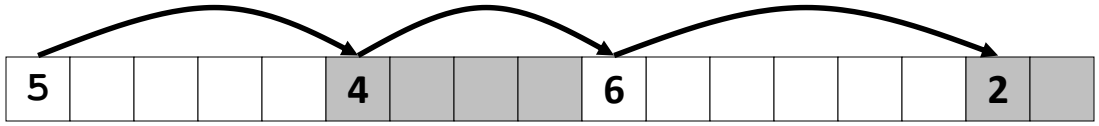
第9章 虚拟内存：
动态内存分配 ——高级概念

主要内容

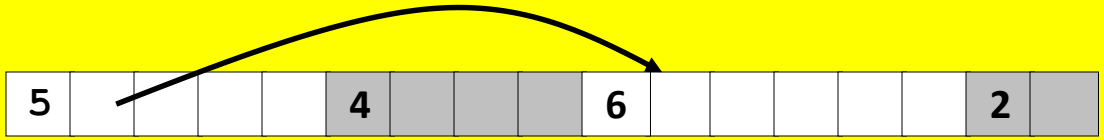
- **Explicit free lists** 显式空闲链表
- Segregated free lists 分离的空闲链表
- Garbage collection 垃圾收集
- Memory-related perils and pitfalls内存相关的风险和陷阱

Keeping Track of Free Blocks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

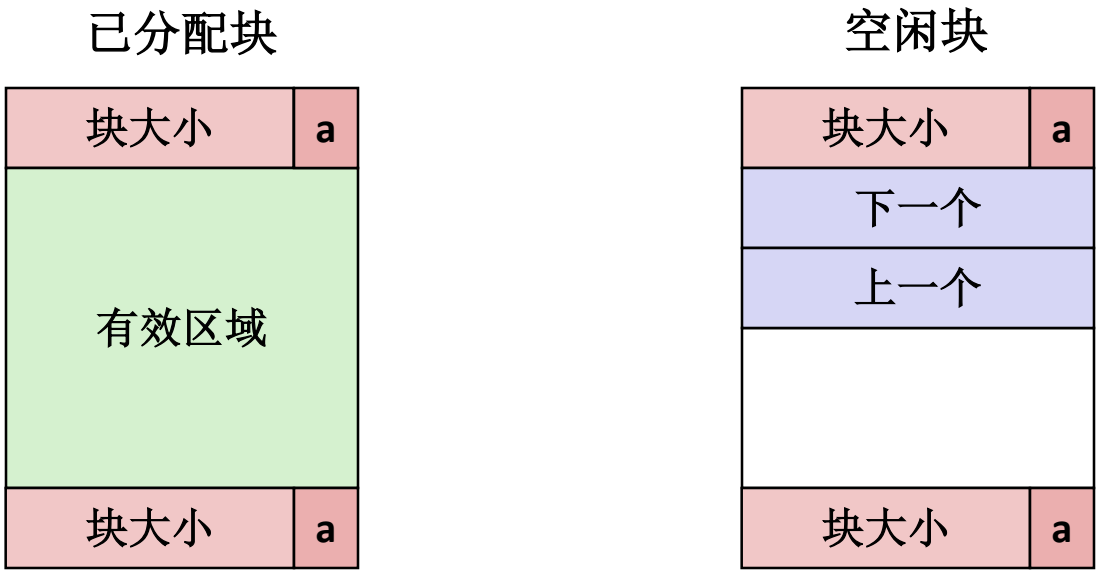


- 方法 2: *显式空闲链表* 在空闲块中使用指针连接空闲块



- 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。

Explicit Free Lists显式空闲链表

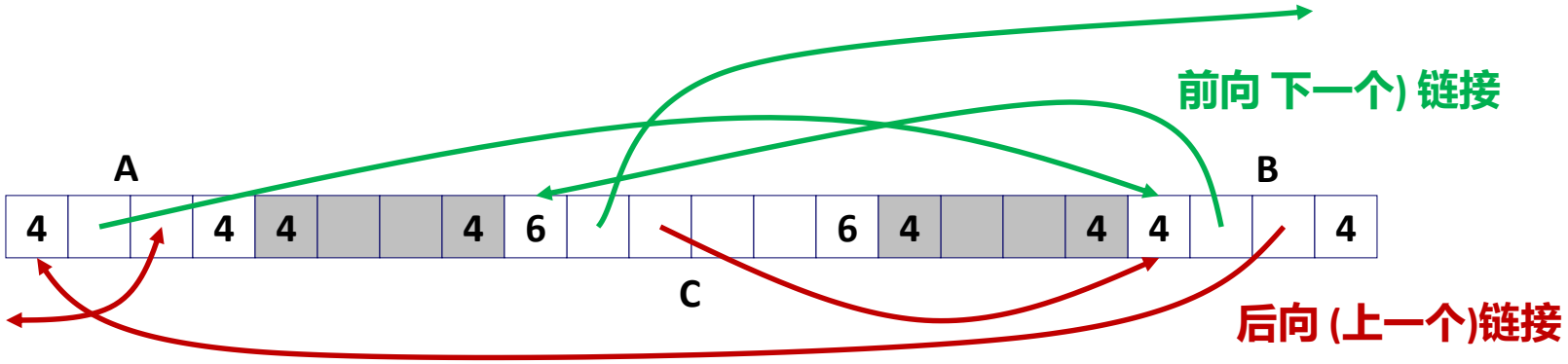


- 保留*空闲块*链表，而不是*所有块*
  - “下一个”空闲块可以在任何地方
    - 因此我们需要存储前/后指针，而不仅仅是大小
  - 还需要合并边界标记
  - 幸运的是，我们只跟踪空闲块，所以我们可以使用有效区域。

Explicit Free Lists显式空闲链表

- 逻辑地：

- 物理地: 块的顺序是任何的

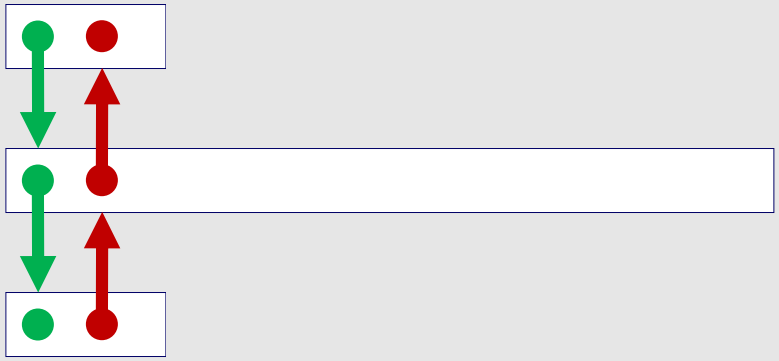


Allocating From Explicit Free Lists

显式空闲链表的分配

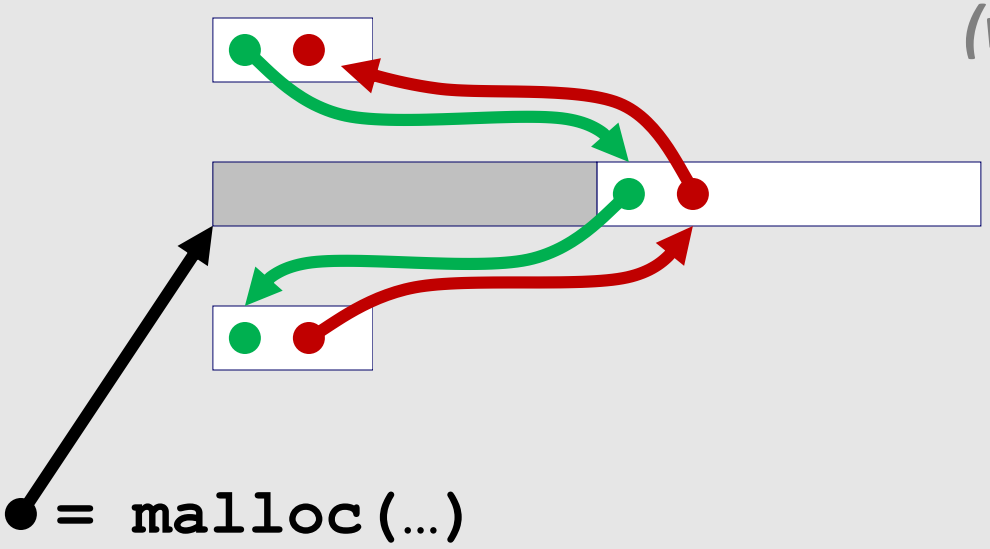
分配示意图

分配前



分配后

(with splitting)



Freeing With Explicit Free Lists
显式空闲链表释放

- *插入原则*: **Where in the free list do you put a newly freed block?一个新释放的块放在空闲链表的什么位置？**
- **LIFO (last-in-first-out) policy后进先出法**
  - 将新释放的块放置在链表的开始处
  - *Pro赞成:* 简单，常数时间
  - *Con反对:* 研究表明碎片比地址排序更糟糕

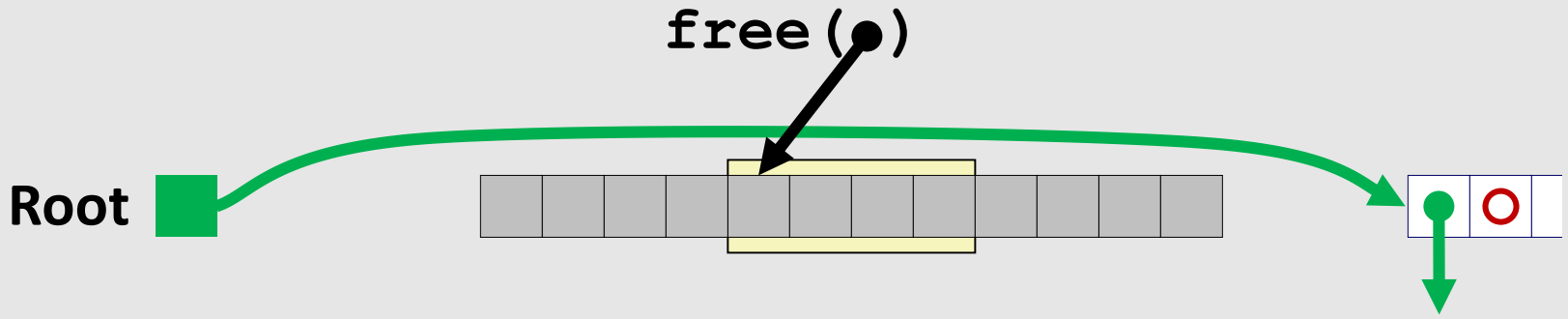
- **Address-ordered policy地址顺序法**
  - 按照地址顺序维护链表:

    addr(祖先) < addr(当前回收块) < addr(后继)
  - *Con反对:* 需要搜索
  - *Pro赞成:* 研究表明碎片要少于LIFO (后进先出法)

LIFO (后进先出) 的回收策略 (案例1)

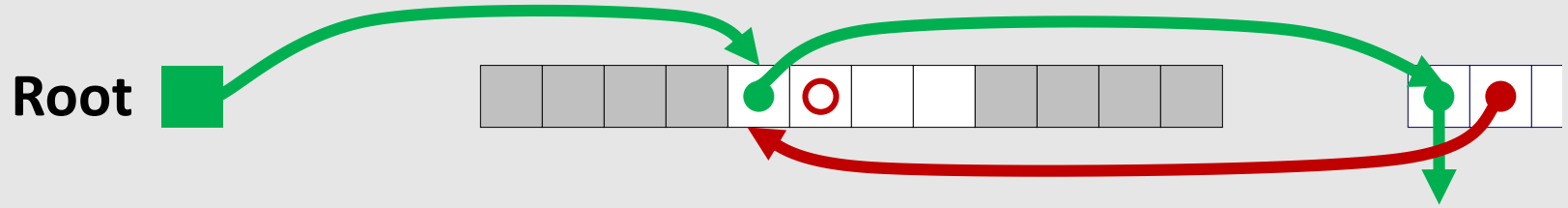
conceptual graphic

回收前



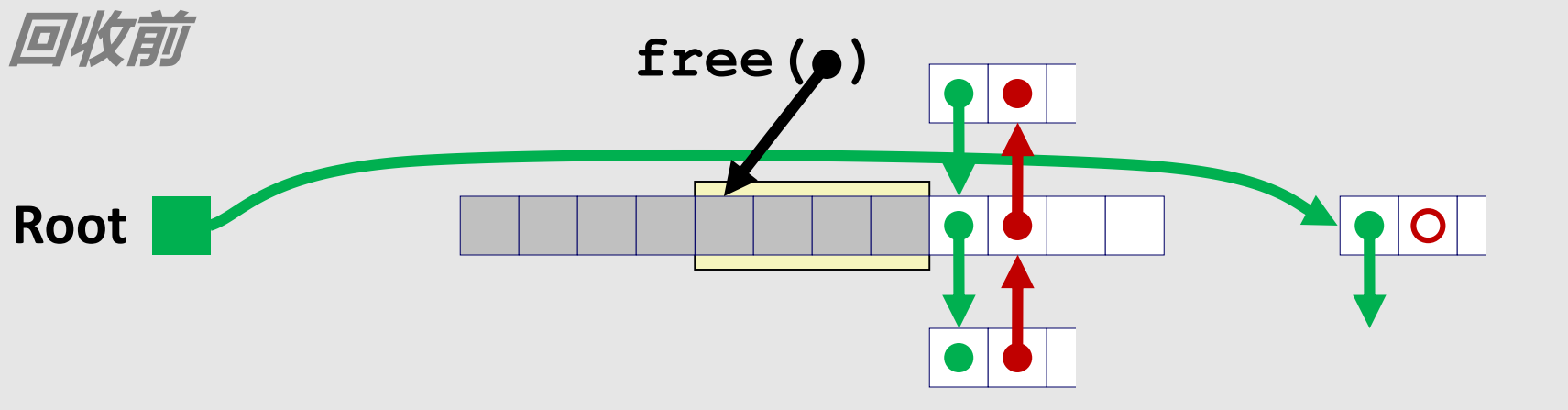
- 将新释放的块放置在链表的开始处

回收后

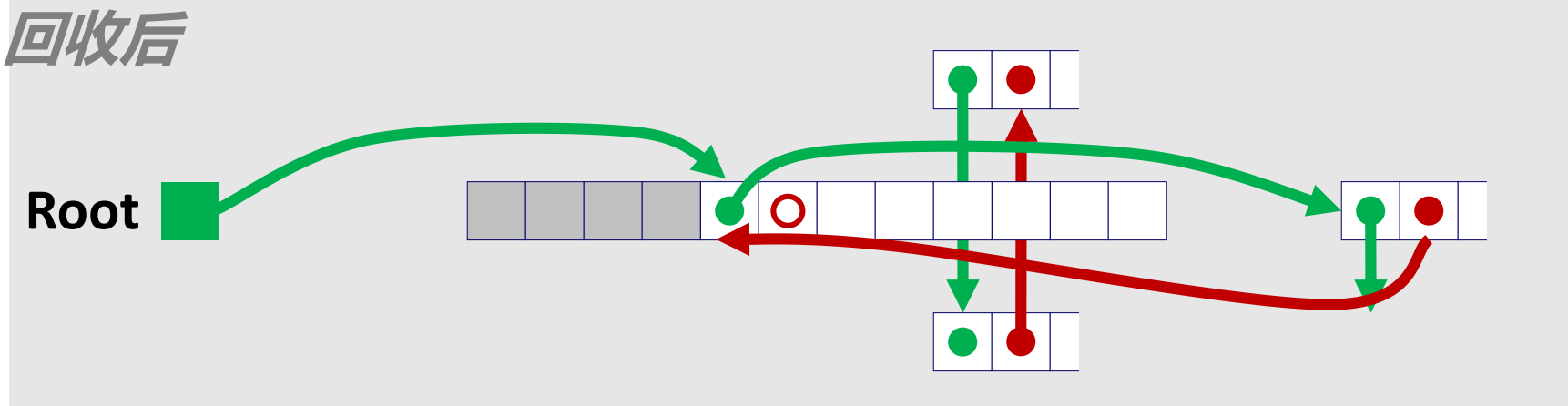


LIFO (后进先出) 的回收策略 (案例2)

conceptual graphic

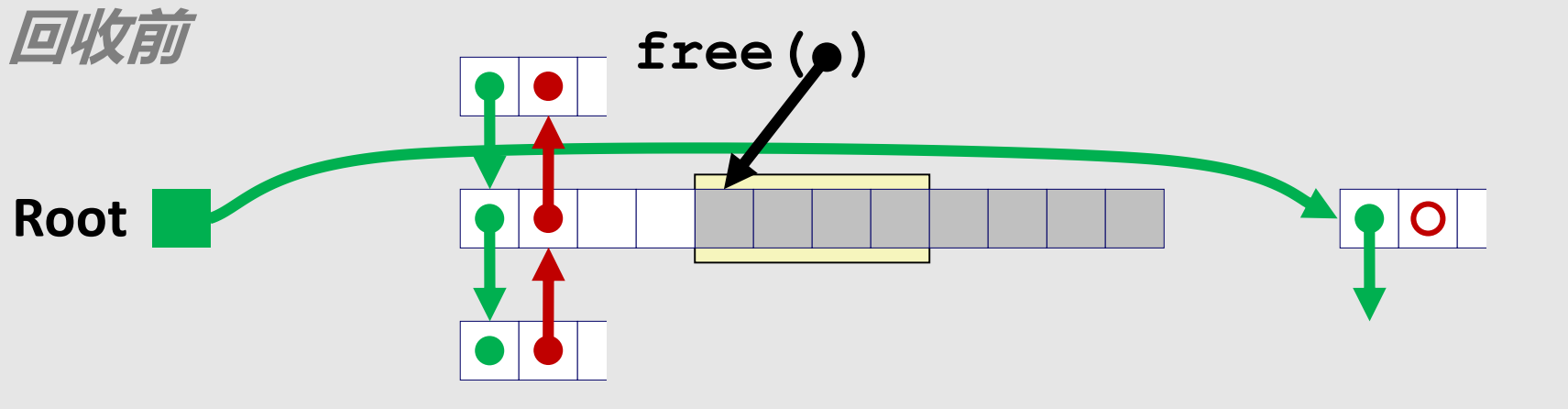


- 将后继块拼接出来，合并两个内存块，并在列表的开始处插入新块

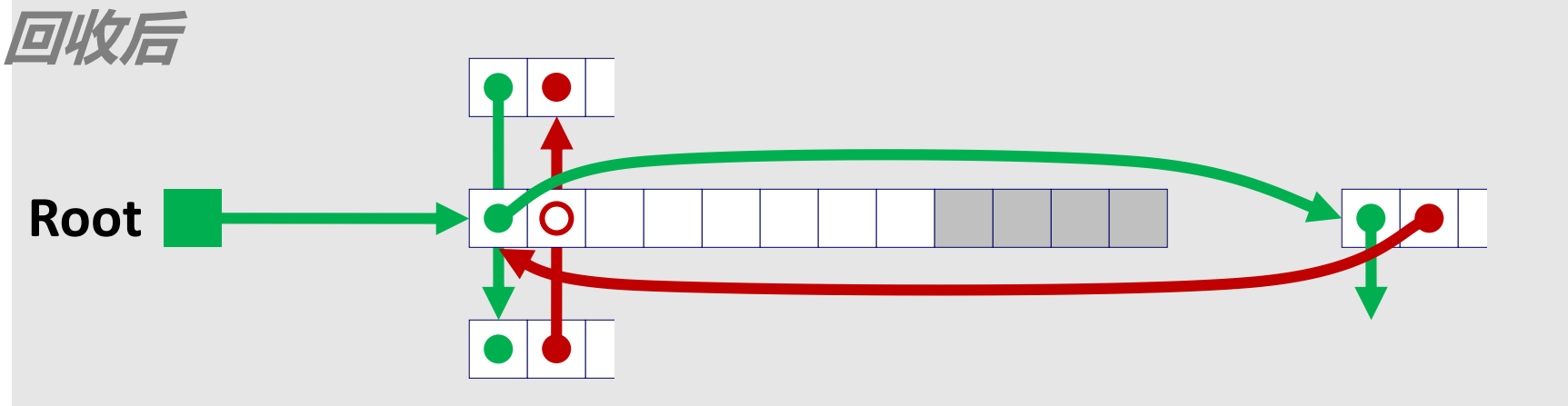


LIFO (后进先出) 的回收策略 (案例3)

conceptual graphic

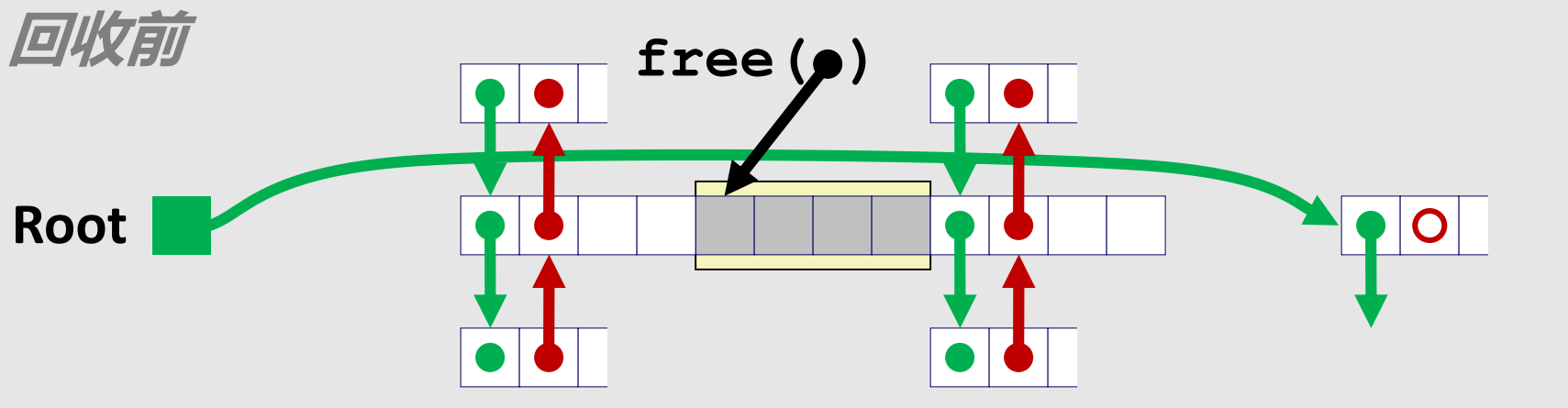


- 拼接出前块，合并两个内存块，并在列表的开始处插入新的块

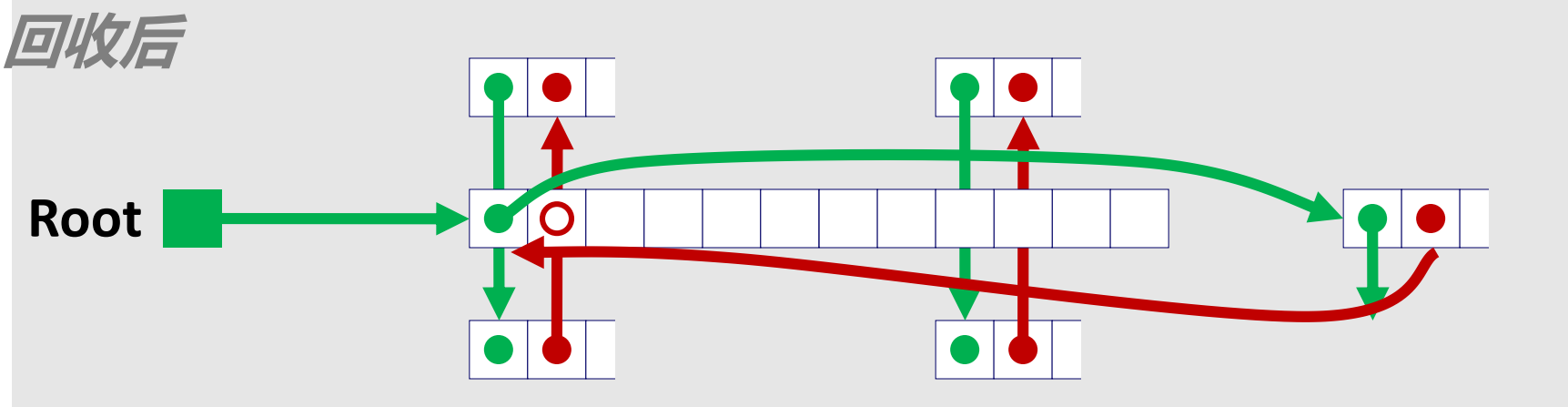


LIFO (后进先出) 的回收策略 (案例4)

conceptual graphic



- 将前块和后继块拼接起来，将所有3个内存块合并并在列表的开始处插入新块

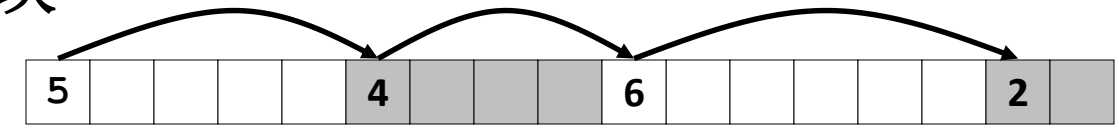


Explicit List Summary（显式链表小结）

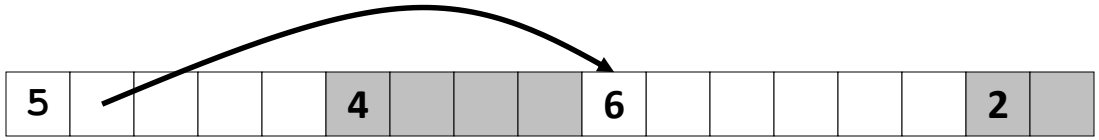
- 与隐式链表相比较:
  - 分配时间从块总数的线性时间减少到空闲块数量的线性时间
    - 当大量内存被占用时 *快得多Much faster*
  - 因为需要在列表中拼接块，释放和分配稍显复杂一些
- 一些*特殊的?额外的?(*extra) space for the links (2 extra words needed for each block)每个块需要两个额外的字
  - 这样会不会增加内部碎片？
- 最常用的链表连接是将分离的空闲链表结合在一起
  - 维护多个不同大小类的或者有可能的话维护多个不同对象类型的链表

Keeping Track of Free Blocks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



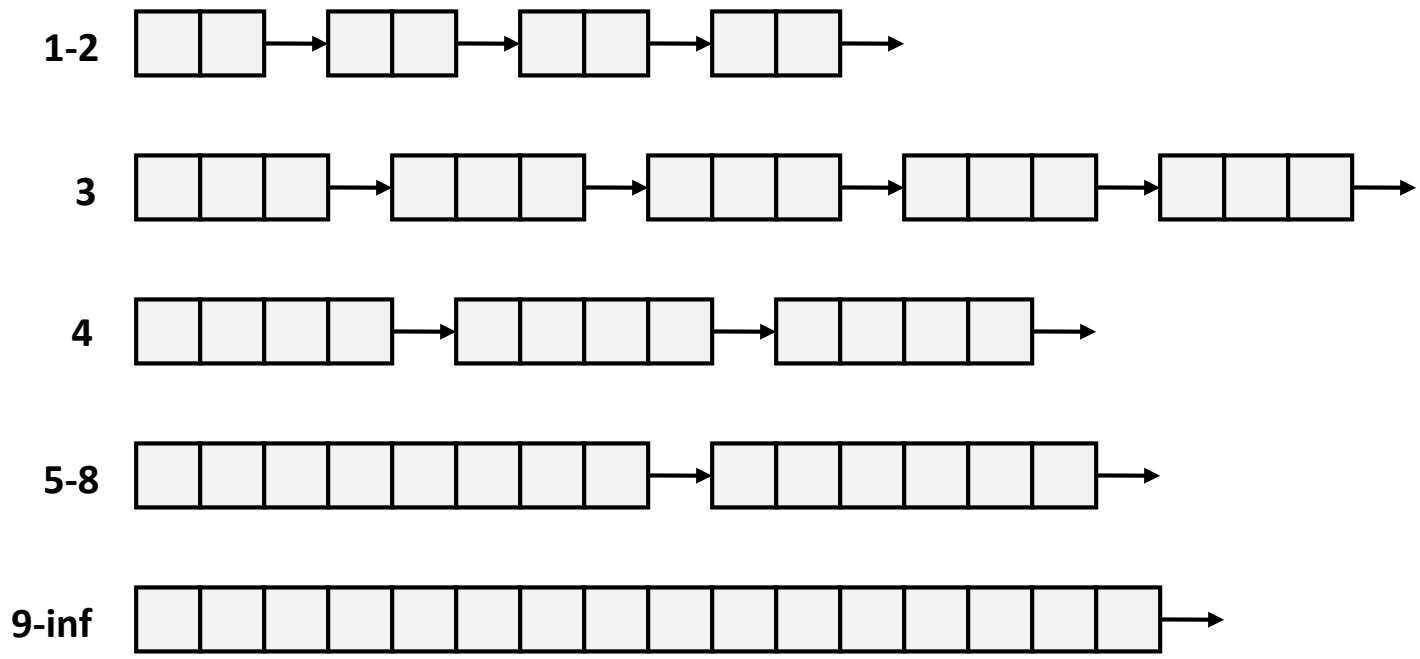
- 方法 3: *分离的空闲链表*
  - 每个大小类的空闲链表包含大小相等的块
- 方法 4: *按照尺寸排序的块*
  - 可以使用平衡树（例如红黑树），在每个空闲块中有指针，尺寸作为键。

主要内容

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

Segregated List (Seglist) Allocators 分离链表分配器

- 每个*大小类*中的块构成一个空闲链表



- 通常每个小块都有单独的大小类
- 对于大块: 按照2的幂分类

Seglist Allocator 分离适配

- 分配器维护空闲链表数组，每个大小类一个空闲链表
- 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.

Seglist Allocator (cont.) 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.
    - 极端示例：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。

More Info on Allocators 关于分配的更多信息

- **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
  - 从 CS:APP student site (csapp.cs.cmu.edu) 可以获取

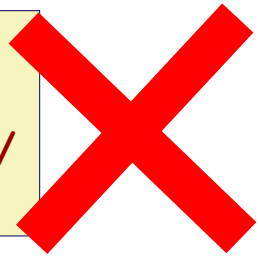
主要内容

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

Implicit Memory Management: Garbage Collection
隐式内存管理----垃圾收集

- *垃圾收集:* 自动回收堆存储的过程—应用从来不显式释放

```
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

- **“保守的”垃圾收集器为 C 和 C++ 程序提供垃圾收集**
  - 然而，它并不能收集所有的垃圾

Garbage Collection 垃圾收集

- 内存管理器如何知道何时可以释放内存？
  - 一般我们不知道下一步会用到什么，因为这取决于具体条件
  - 但是我们知道如果没有指针，某些块就不能被使用

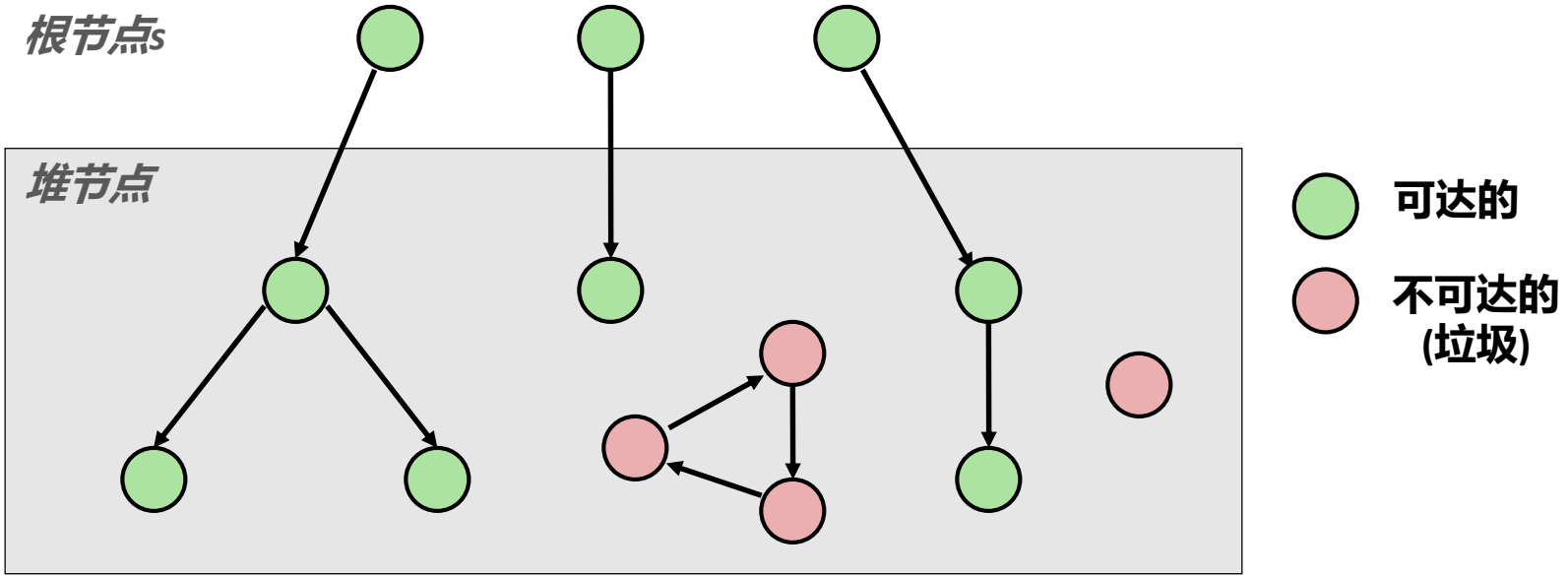
- 必须做些关于指针的假设
  - 内存管理器可以区分指针和非指针
  - 所有指针都指向一个块的起始地址
  - 无法隐藏指针
    (e.g., by coercing them to an `int`, and then back again)

Classical GC Algorithms 经典的垃圾收集算法

- **Mark-and-sweep collection (McCarthy, 1960) 标记-清除 算法**
  - 不移动块 (除非要 “紧凑”)
- **Reference counting (Collins, 1960) 引用计数**
  - 不移动块 (未讨论)
- **Copying collection (Minsky, 1963) 复制收集**
  - 移动块 (未讨论)
- **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.

Memory as a Graph将内存当作图

- 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）

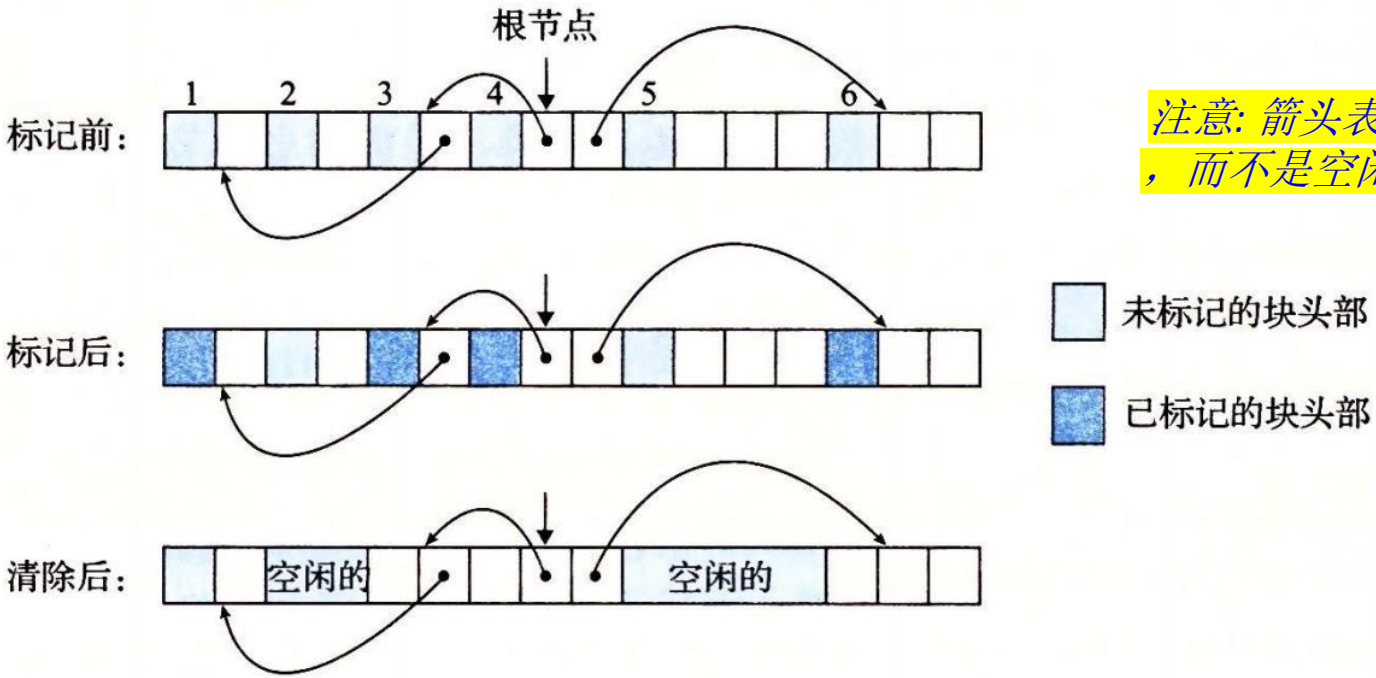


可达节点： 存在一条从根节点出发并到达该节点的有向路径

不可达节点是垃圾(不能被应用再次使用)

Mark and Sweep Collecting 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分配直到你“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。

Assumptions For a Simple Implementation
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以字为单位的长度（不包括头部）
  - `get_roots()`: 返回所有根节点

Mark and Sweep (cont.)

Mark（标记） 使用内存图的深度优先遍历

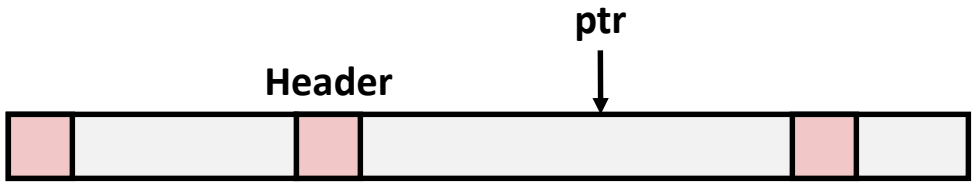
```
ptr mark(ptr p) {
    if (!is_ptr(p)) return;          // 不是指针则什么都不做
    if (markBitSet(p)) return;       // 检查是否已标记
    setMarkBit(p);                   // 设置标记位
    for (i=0; i < length(p); i++)    // 调用mark标记块中
      mark(p[i]);                    // 的每个字
    return;
}
```

Sweep （清除） 使用长度查找下一个块

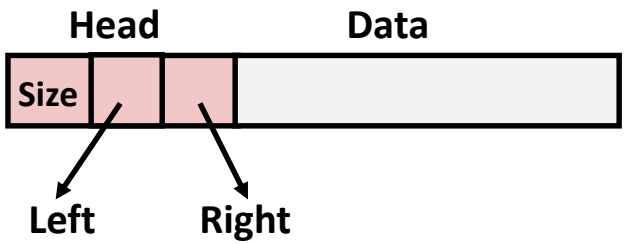
```
ptr sweep(ptr p, ptr end) {
    while (p < end) {
        if markBitSet(p)
            clearMarkBit();
        else if (allocateBitSet(p))
            free(p);
        p += length(p);
}
```

C程序的保守的Mark & Sweep

- C程序的“保守的”垃圾收集器
  - `is_ptr()` determines if a word is a pointer by checking if it points to an allocated block of memory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（遍历二叉树找到节点left<ptr<right）
  - 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）



Left: 较小的地址
Right: 较大的地址

主要内容

- Explicit free lists 显示空闲链表
- Segregated free lists 分离的空闲链表
- Garbage collection 垃圾收集
- **Memory-related perils and pitfalls 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

- Dereferencing bad pointers 不要引用坏指针
- Reading uninitialized memory 读未初始化的内存
- Overwriting memory 覆盖内存
- Referencing nonexistent variables 引用不存在的变量
- Freeing blocks multiple times 多次释放内存
- Referencing freed blocks 引用空闲堆块中的数据
- Failing to free blocks 释放内存失败

Dereferencing Bad Pointers间接引用坏指针

- 经典的scanf错误

```
int val;

...

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

正确版本：`scanf("%d", &val);`

Reading Uninitialized Memory
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;
}
```

正确版本： `int *y = calloc(N, sizeof(int));`

calloc在创建动态数组时默认清零

Overwriting Memory 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

正确版本： `p = malloc(N*sizeof(int*));`

Overwriting Memory 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++) {   //i<n
   p[i] = malloc(M*sizeof(int));
}
```

正确版本： for (i=0; i<N; i++) { //i<n

Overwriting Memory 覆盖内存

- Not checking the max string size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

正确版本：fgets(s, size, stdin);

Overwriting Memory覆盖内存

- Misunderstanding pointer arithmetic误解指针运算

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

   while (*p && *p != val)
      p += sizeof(int);                  //+16

   return p;
}
```

正确版本：p +=1;

Overwriting Memory覆盖内存

- **Referencing a pointer instead of the object it points to**
  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);
}
```

正确版本：`(*size)--;`

对于大顶堆和小顶堆删除操作，如果堆中元素用数组存放，需要在取出堆顶元素后，用最后一个元素放在第一个位置上，重新调整堆结构。正常这个函数第二个参数应该是个堆元素个数，结果用了指针传入，所以我们需要修改指针指向的内容，即堆元素总个数减1。

Referencing Nonexistent Variables
引用不存在的变量

- Forgetting that local variables disappear when a function returns忘记当函数返回时局部变量将消失

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

    return &val;
}
```

函数本身错在不应该返回即将消失的局部变量val的地址

Freeing Blocks Multiple Times多次释放

- Nasty!很讨厌

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

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

正确版本：free(y)
malloc和free应该成对出现，所以不应该两次free(x)但没有free(y)

Referencing Freed Blocks引用空闲堆块中的数据

- Evil!很邪恶

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

X指向的内存空间已经被释放掉了，错在引用野指针

Failing to Free Blocks (Memory Leaks)
释放失败（内存泄漏）

- Slow, long-term killer!

```
foo() {
    int *x = malloc(N*sizeof(int));
    ...
    return;                          //x是垃圾，应释放
}
```

正确版本：return前面加一条语句 free(x)
malloc和free应该成对出现

Failing to Free Blocks (Memory Leaks)
释放失败（内存泄漏）

- Freeing only part of a data structure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;
}
```

正确版本：增加多个free(节点指针)

本程序原意是后续创建了很多个节点，但都没有free，只free了一个头节点

Dealing With Memory Bugs 处理内存bugs

- **Debugger: `gdb`**
  - Good for finding bad pointer dereferences有利于发现间接引用坏指针
  - Hard to detect the other memory bugs很难检测其他内存bug
- **Data structure consistency checker数据一致性检查**
  - Runs silently, prints message only on error安静运行，仅在错误时打印消息
  - Use as a probe to zero in on error用作探针以瞄准错误
- **Binary translator: `valgrind` 二进制转换器**
  - Powerful debugging and analysis technique强大的调试和分析技术
  - Rewrites text section of executable object file重写可执行目标文件的文本段
  - Checks each individual reference at runtime运行时检查每个引用
    - Bad pointers, overwrites, refs outside of allocated block坏指针，覆盖写，引用已分配块之外的内容
- **glibc malloc contains checking code**
  - `setenv MALLOC_CHECK_ 3`

Hope you enjoyed the CSAPP course!