Dynamic Memory Allocation: Advanced Concepts
动态存储分配:高级概念

100076202： 计算机系统导论

任课教师：

计卫星　　宿红毅　　张艳

原作者：

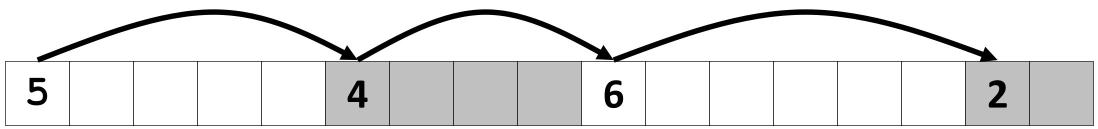
Randal E. Bryant and David R. O'Hallaron

内容提纲/Today

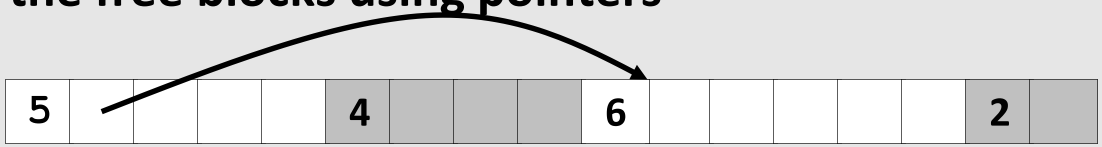
- 显示空闲列表/Explicit free lists
- 分离的空闲列表/Segregated free lists
- 垃圾收集/Garbage collection
- 内存相关的风险和陷阱/Memory-related perils and pitfalls

跟踪空闲块/Keeping Track of Free Blocks

- **方法1：隐式空闲列表使用长度链接所有块/Method 1: *Implicit free list* using length—links all blocks**



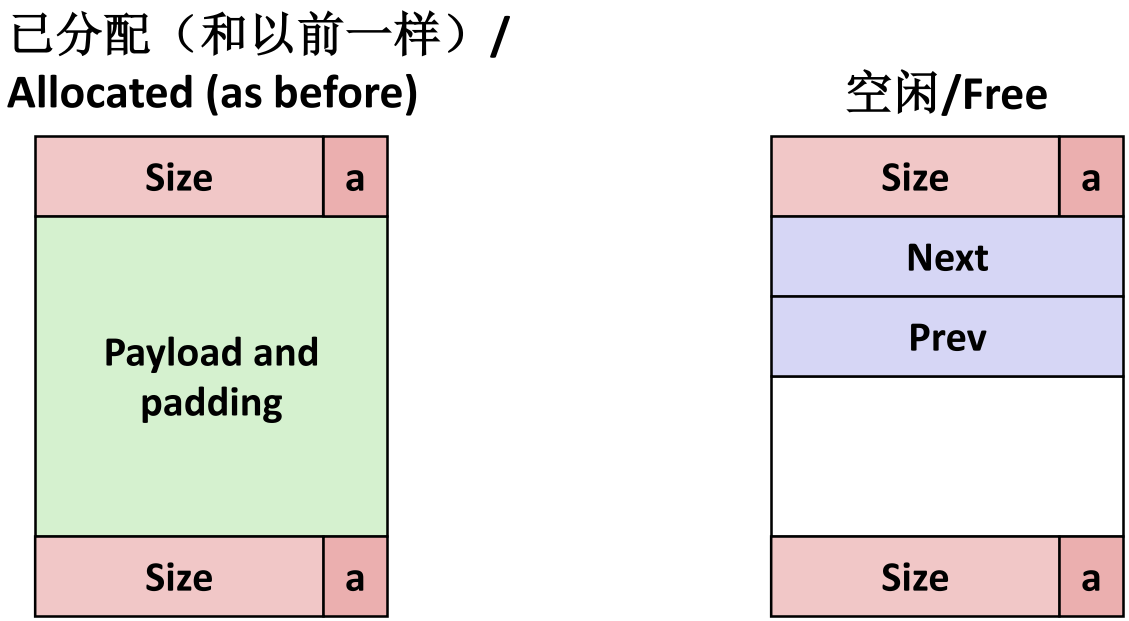
- **方法2：显式空闲列表使用指针串接空闲块/Method 2: *Explicit free list* among the free blocks using pointers**



- **方法3：分离的空闲列表/Method 3: *Segregated free list***
  - 不同大小的块使用不同的列表管理/Different free lists for different size classes

- **方法4：根据大小排序块/Method 4: *Blocks sorted by size***
  - 使用平衡红黑树/Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

显式空闲列表/Explicit Free Lists

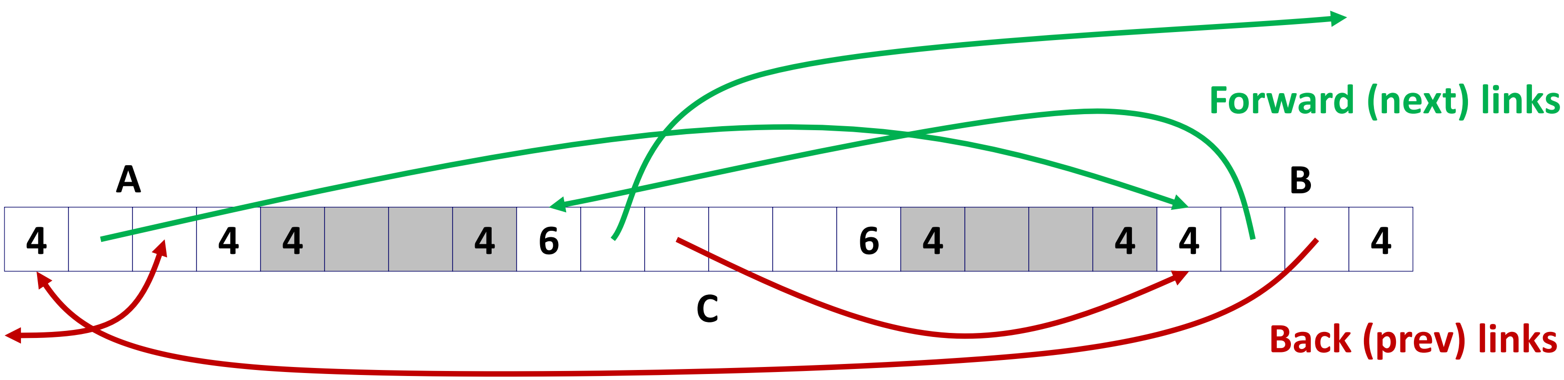


- 维护空闲块列表，而不是所有块/**Maintain list(s) of *free* blocks, not *all* blocks**
  - 下一个空闲块可能在任一地方/The "next" free block could be anywhere
    - 所以需要前向/后向指针，不只是大小/So we need to store forward/back pointers, not just sizes
  - 仍然需要使用边界标记进行合并/Still need boundary tags for coalescing
  - 幸运地是我们只需要跟踪空闲块，所以可以使用载荷区域/Luckily we track only free blocks, so we can use payload area

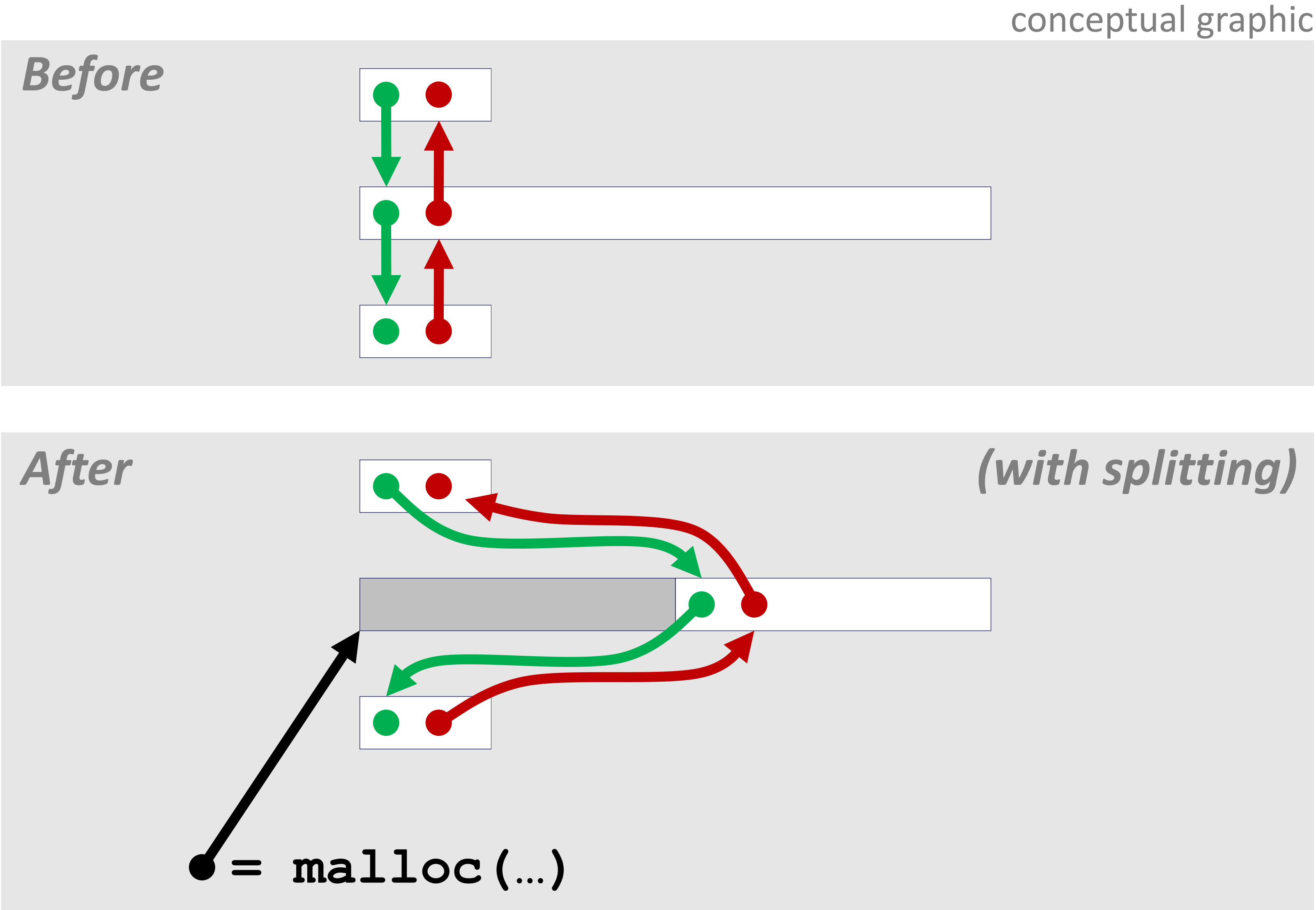
显式空闲列表/Explicit Free Lists

- 逻辑上/Logically:

- 物理上：块可能是任意顺序/Physically: blocks can be in any order



从显式空闲列表分配/Allocating From Explicit Free Lists



释放空闲块到显式空闲列表/Freeing With Explicit Free Lists

- ***插入策略:*** 在空闲列表的什么位置插入一个新的空闲块？/***Insertion policy*: Where in the free list do you put a newly freed block?**
- **LIFO (last-in-first-out) policy**
  - 在空闲列表的开始插入空闲块/Insert freed block at the beginning of the free list
  - ***优点:*** 简单并且常数时间完成/***Pro:*** simple and constant time
  - ***缺点:*** 研究表明比地址排序导致更多的碎片/***Con:*** studies suggest fragmentation is worse than address ordered

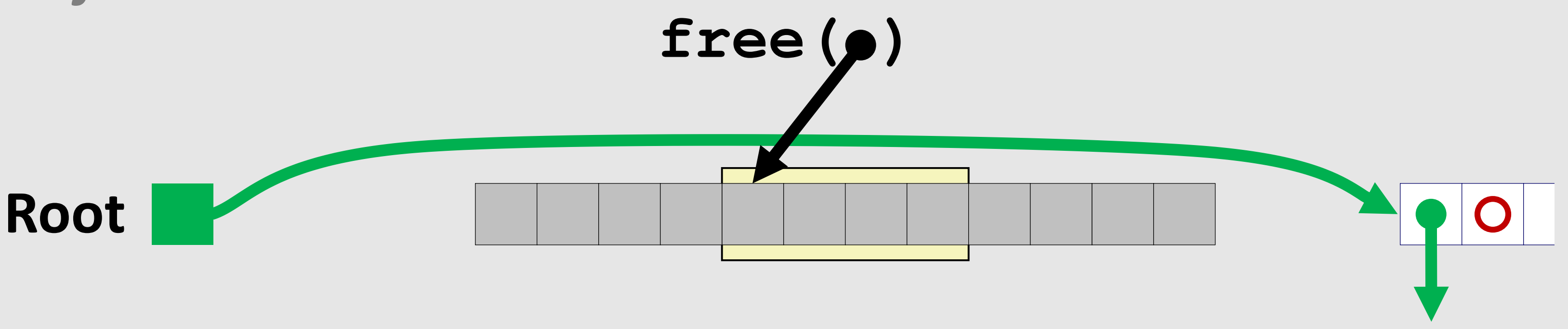
- **地址排序策略/Address-ordered policy**
  - 插入空闲块后列表中的地址总是排序的/Insert freed blocks so that free list blocks are always in address order:

    addr(prev) < addr(curr) < addr(next)
  - ***Con:*** requires search/缺点：需要搜索
  - ***Pro:*** studies suggest fragmentation is lower than LIFO/优点：研究表明比LIFO有低的内存碎片

基于LIFO策略的释放/Freeing With a LIFO Policy (Case 1)

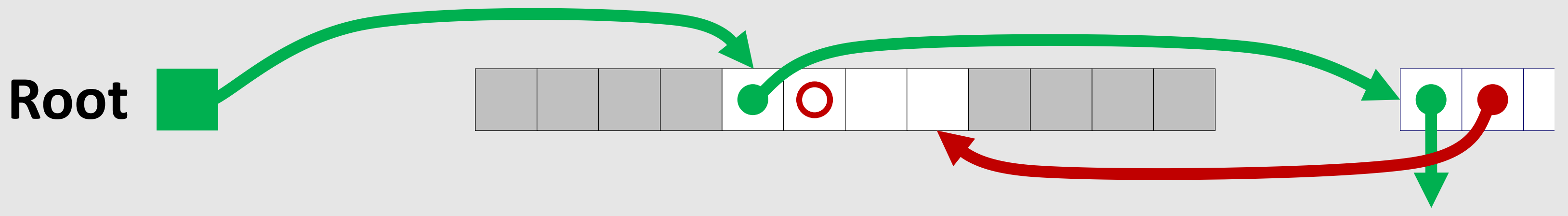
conceptual graphic

Before



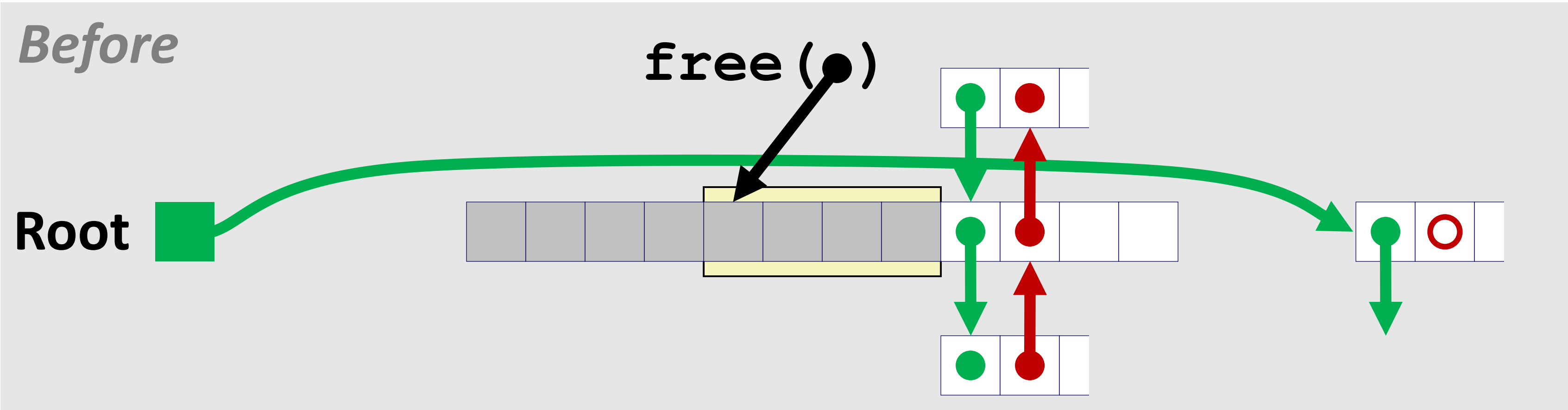
- 将空闲块插入到列表头/Insert the freed block at the root of the list

After

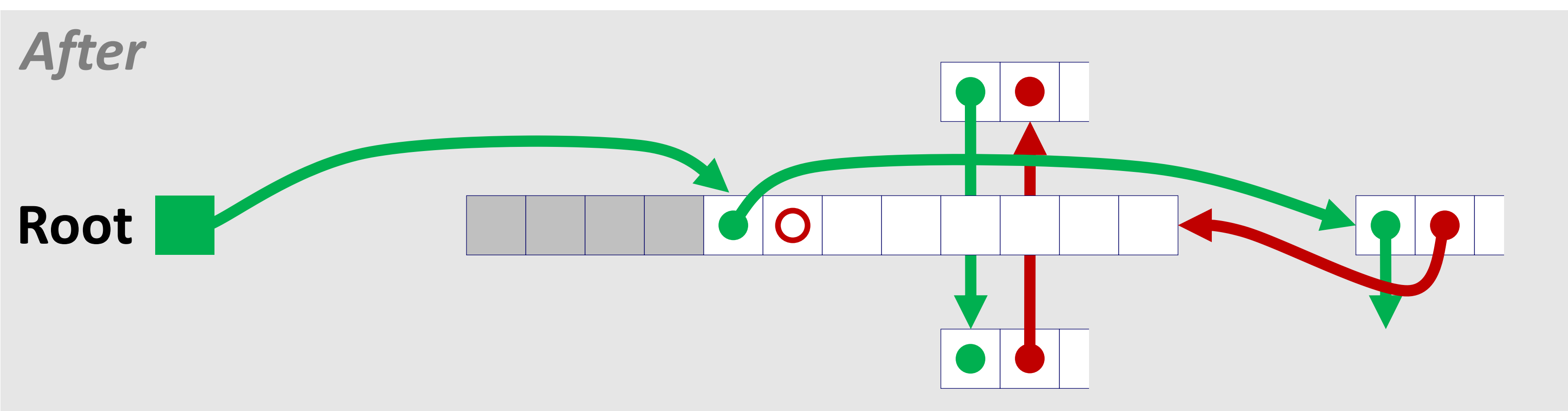


基于LIFO策略的释放/ Freeing With a LIFO Policy (Case 2)

conceptual graphic

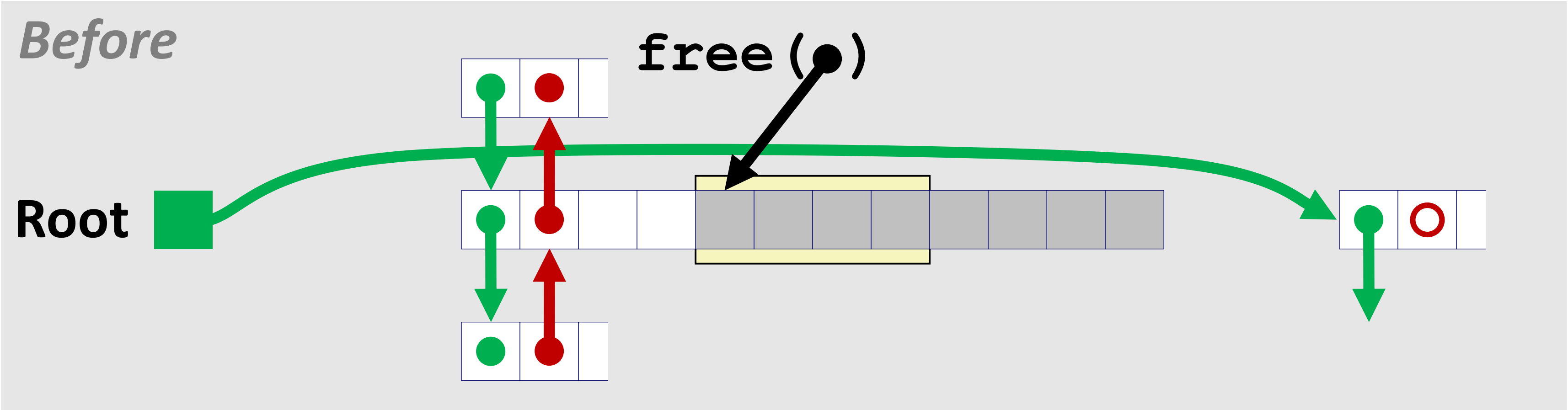


- 拼接出后续块，合并两个块并在列表头插入新块/Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list

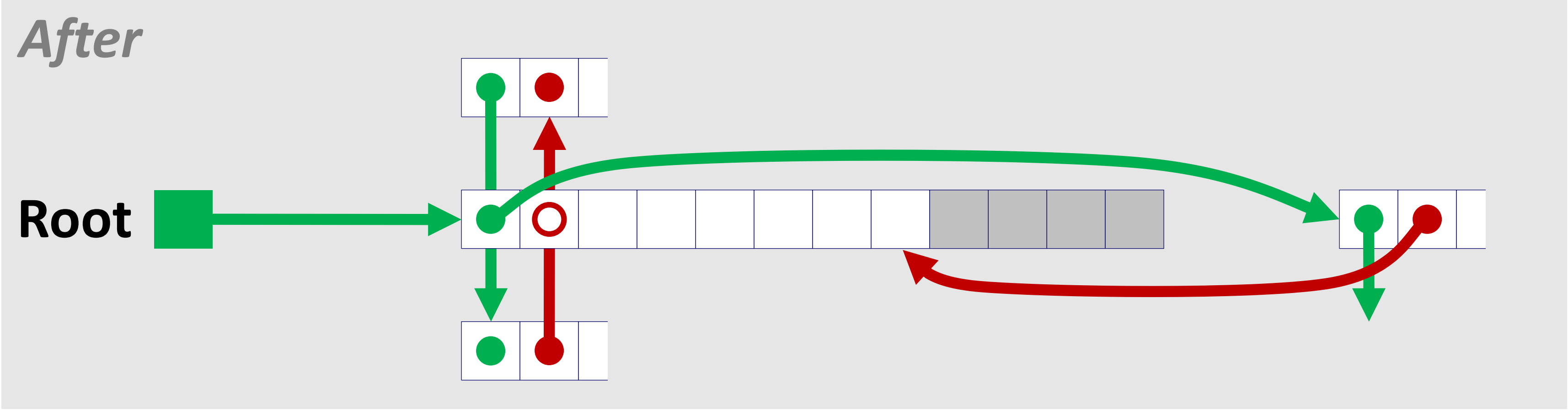


基于LIFO策略的释放/ Freeing With a LIFO Policy (Case 3)

conceptual graphic

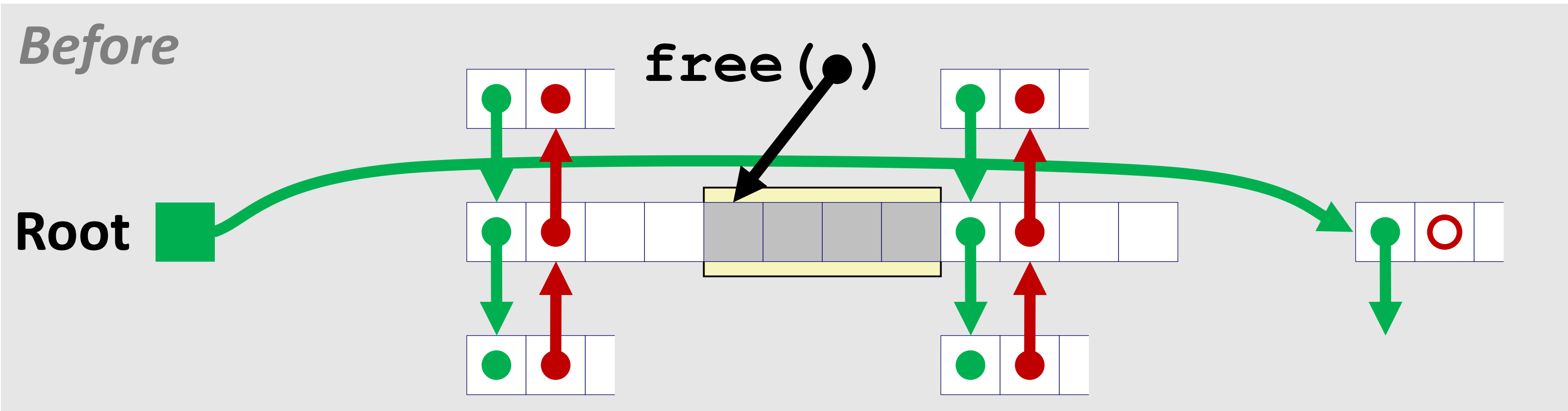


- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list

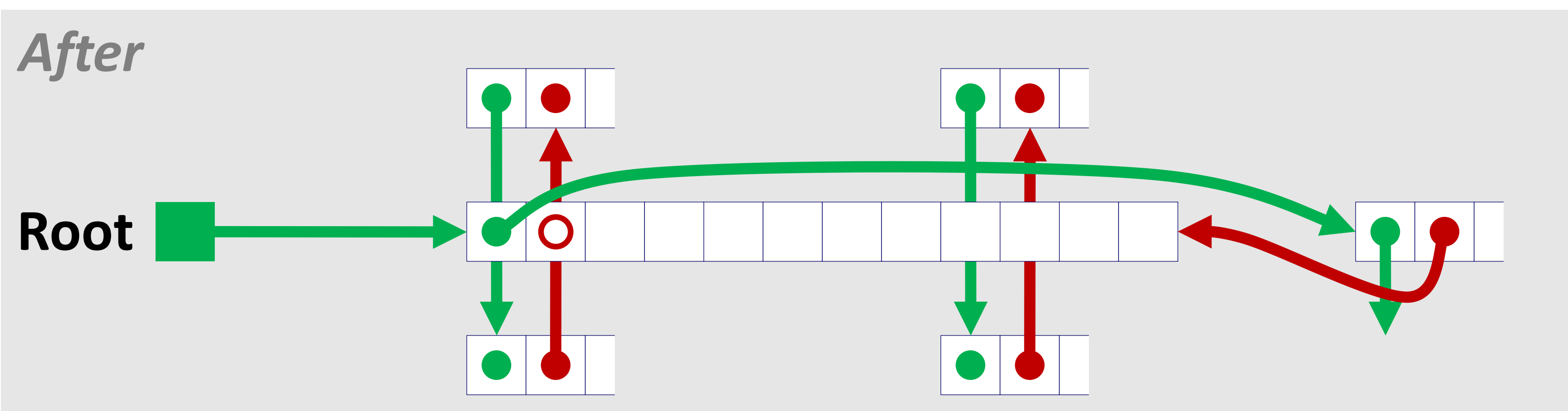


基于LIFO策略的释放/ Freeing With a LIFO Policy (Case 4)

conceptual graphic



- Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new block at the root of the list

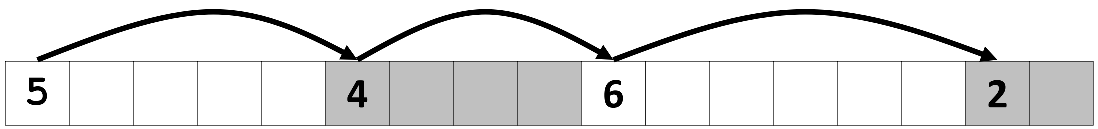


显式列表总结/Explicit List Summary

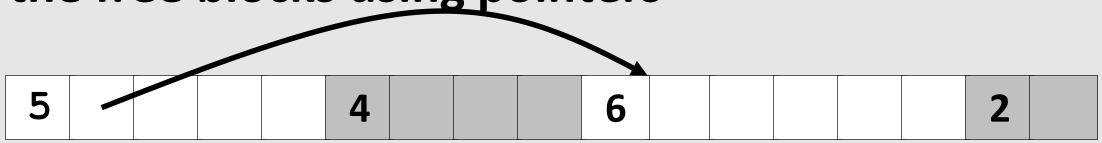
- **与隐式列表相比/Comparison to implicit list:**
  - 分配时间与空闲块的数量成线性时间，而不是所有的块/Allocate is linear time in number of ***free*** blocks instead of ***all*** blocks
    - 当内存大部分被占用的时候快很多/***Much faster*** when most of the memory is full
  - 由于从列表中删除和向链表中插入块，分配和释放稍微复杂一些/Slightly more complicated allocate and free since needs to splice blocks in and out of the list
  - 链接需要一些额外的空间/Some extra space for the links (2 extra words needed for each block)
    - 会增加内部碎片吗？/Does this increase internal fragmentation?
- **链表通常是和分离的空闲列表一起使用的/Most common use of linked lists is in conjunction with segregated free lists**
  - 保持多个不同大小类的列表，或者为不同的对象设置不同的列表/Keep multiple linked lists of different size classes, or possibly for different types of objects

跟踪空闲块/Keeping Track of Free Blocks

- 方法1：隐式列表使用长度链接所有块/Method 1: *Implicit free list* using length—links all blocks



- 方法2：显式空闲列表使用指针串接空闲块/Method 2: *Explicit free list* among the free blocks using pointers



- 方法3：分离的空闲列表/Method 3: *Segregated free list*
  - 不同大小的块使用不同的链表管理/Different free lists for different size classes

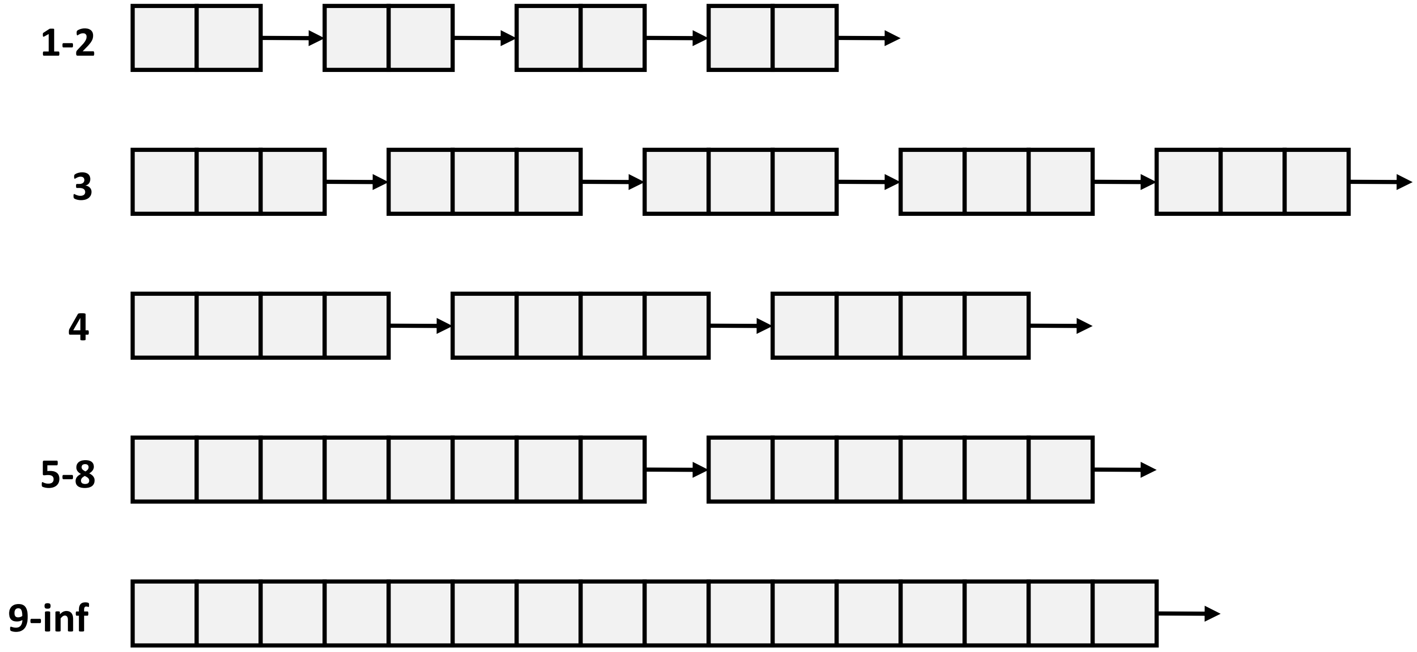
- 方法4：根据大小排序块/Method 4: *Blocks sorted by size*
  - 使用平衡红黑树/Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

内容提纲/Today

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

分离空闲列表分配器/Segregated List (Seglist) Allocators

- 每个不同大小类块有自己的空闲列表/Each *size class* of blocks has its own free list



- 通常比较小的块有自己单独的类/Often have separate classes for each small size
- 对于比较大的块，每个2的指数区间有一个类/For larger sizes: One class for each two-power size

Seglist分配器/Seglist Allocator

- 空闲列表数组中的每个元素对应某个大小类，**Given an array of free lists, each one for some size class**

- 分配大小为n的块时：**/To allocate a block of size *n*:**
  - 搜索对应的空闲列表，其中的块大小m> n/Search appropriate free list for block of size *m > n*
  - 如果找到一个合适的块：/If an appropriate block is found:
    - 拆分块并将碎片挂接到对应的列表（可选）/Split block and place fragment on appropriate list (optional)
  - 如果没找到，则尝试下一个更大的列表/If no block is found, try next larger class
  - 重复以上步骤直到找到一个块/Repeat until block is found

- 如果没找到：**/If no block is found:**
  - 从OS申请更多的堆内存/Request additional heap memory from OS (using `sbrk()`)
  - 从新申请的内存分配大小为n字节的块/Allocate block of *n* bytes from this new memory
  - 将剩下的当做一个空闲块放到最大的类表中/Place remainder as a single free block in largest size class.

Seglist分配器/Seglist Allocator (cont.)

- **释放一个块：/To free a block:**
  - 合并并放到合适的列表中/Coalesce and place on appropriate list

- **seglist分配器的优点/Advantages of seglist allocators**
  - 高吞吐率/Higher throughput
    - 对于2的指数次方的大小类是log时间复杂度/log time for power-of-two size classes
  - 更好的内存利用率/Better memory utilization
    - 分离空闲列表中的First-fit 搜索近似于整个堆上的best-fit搜索 /First-fit search of segregated free list approximates a best-fit search of entire heap.
    - 极端案例：/Extreme case: Giving each block its own size class is equivalent to best-fit.

内存分配器的更多资料/More Info on Allocators

- **D. Knuth, "*The Art of Computer Programming*", 2nd edition, Addison Wesley, 1973**
  - The classic reference on dynamic storage allocation

- **Wilson et al, "*Dynamic Storage Allocation: A Survey and Critical Review*", Proc. 1995 Int'l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.**
  - Comprehensive survey
  - Available from CS:APP student site (csapp.cs.cmu.edu)

内容提纲/Today

- 显示空闲列表/Explicit free lists
- 分离的空闲列表/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/Implicit Memory Management: Garbage Collection

- ***垃圾收集***：自动回收堆中分配的内存块-应用程序不用负责释放/***Garbage collection:*** **automatic reclamation of heap-allocated storage—application never has to free**

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

- **许多动态语言的共同特性/Common in many dynamic languages:**
  - Python, Ruby, Java, Perl, ML, Lisp, Mathematica

- **变种（保守的垃圾收集）/Variants (“conservative” garbage collectors) exist for C and C++**
  - 然而，不一定收集所有垃圾/However, cannot necessarily collect all garbage

垃圾收集/Garbage Collection

- 内存管理器如何知道内存什么时候可以被释放？**/How does the memory manager know when memory can be freed?**
  - 通常我们是不知道将来会用到哪些，因为程序执行是有路径分支的/In general we cannot know what is going to be used in the future since it depends on conditionals
  - 但是如果某些块没有指针指向则可以确定是不会用的/But we can tell that certain blocks cannot be used if there are no pointers to them

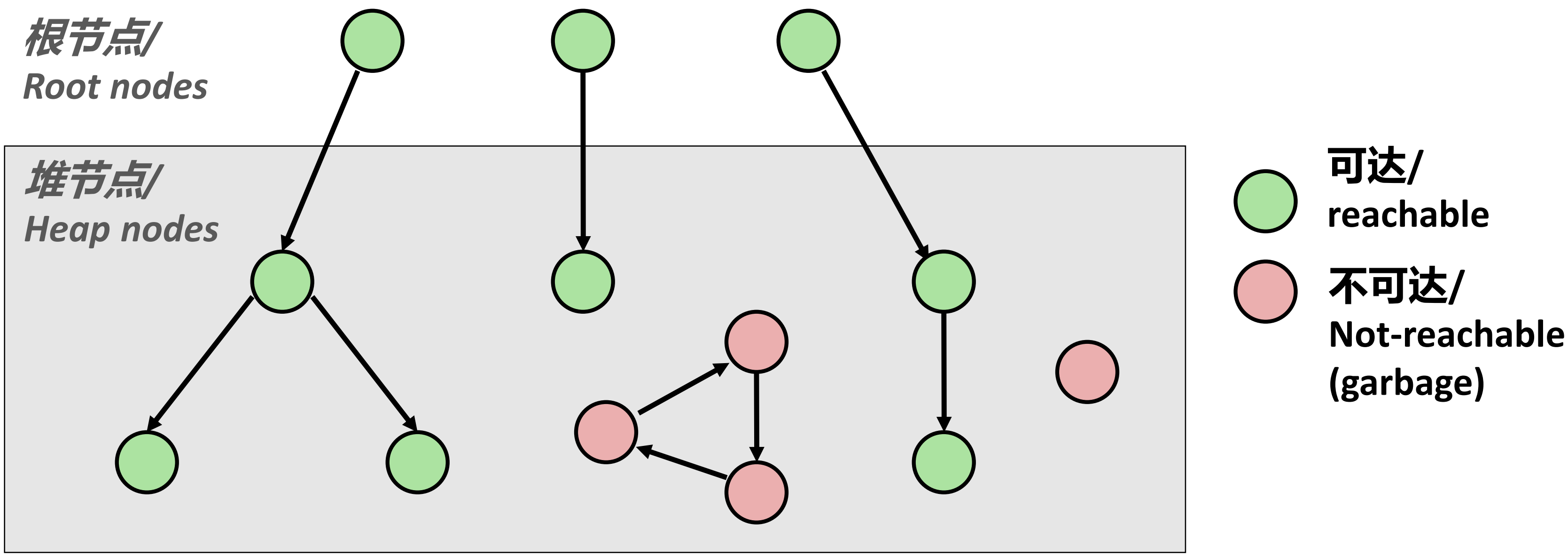
- 关于指针的一些假设**/Must make certain assumptions about pointers**
  - 内存管理器能够区分指针和非指针/Memory manager can distinguish pointers from non-pointers
  - 所有的指针指向块的开始地址/All pointers point to the start of a block
  - 不能隐藏指针/Cannot hide pointers
    (例如，强制转为int，再转回来/e.g., by coercing them to an `int`, and then back again)

经典垃圾收集算法/Classical GC Algorithms

- **标记清除算法/Mark-and-sweep collection (McCarthy, 1960)**
  - 不需要移动内存块（除非需要平移压紧占用部分）/Does not move blocks (unless you also “compact”)
- **引用计数算法/Reference counting (Collins, 1960)**
  - 不需要移动内存块/Does not move blocks (not discussed)
- **拷贝收集算法/Copying collection (Minsky, 1963)**
  - 需要移动内存块/Moves blocks (not discussed)
- **按代垃圾收集算法/Generational Collectors (Lieberman and Hewitt, 1983)**
  - 基于生命周期的收集/Collection based on lifetimes
    - 大部分内存块很快变为垃圾/Most allocations become garbage very soon
    - 主要聚焦在最近分配的区域内开展回收工作/So focus reclamation work on zones of memory recently allocated
- **For more information:**
  Jones and Lin, “*Garbage Collection: Algorithms for Automatic Dynamic Memory*”, John Wiley & Sons, 1996.

将内存当做一个图/Memory as a Graph

- 我们将内存看做一个有向图/**We view memory as a directed graph**
  - 每个块是图中的一个节点/Each block is a node in the graph
  - 每个指针是图中的一条边/Each pointer is an edge in the graph
  - 不在堆中但是持有指向堆中指针的位置称为根节点（例如，寄存器，栈中元素，以及全局变量）/Locations not in the heap that contain pointers into the heap are called ***root*** nodes (e.g. registers, locations on the stack, global variables)

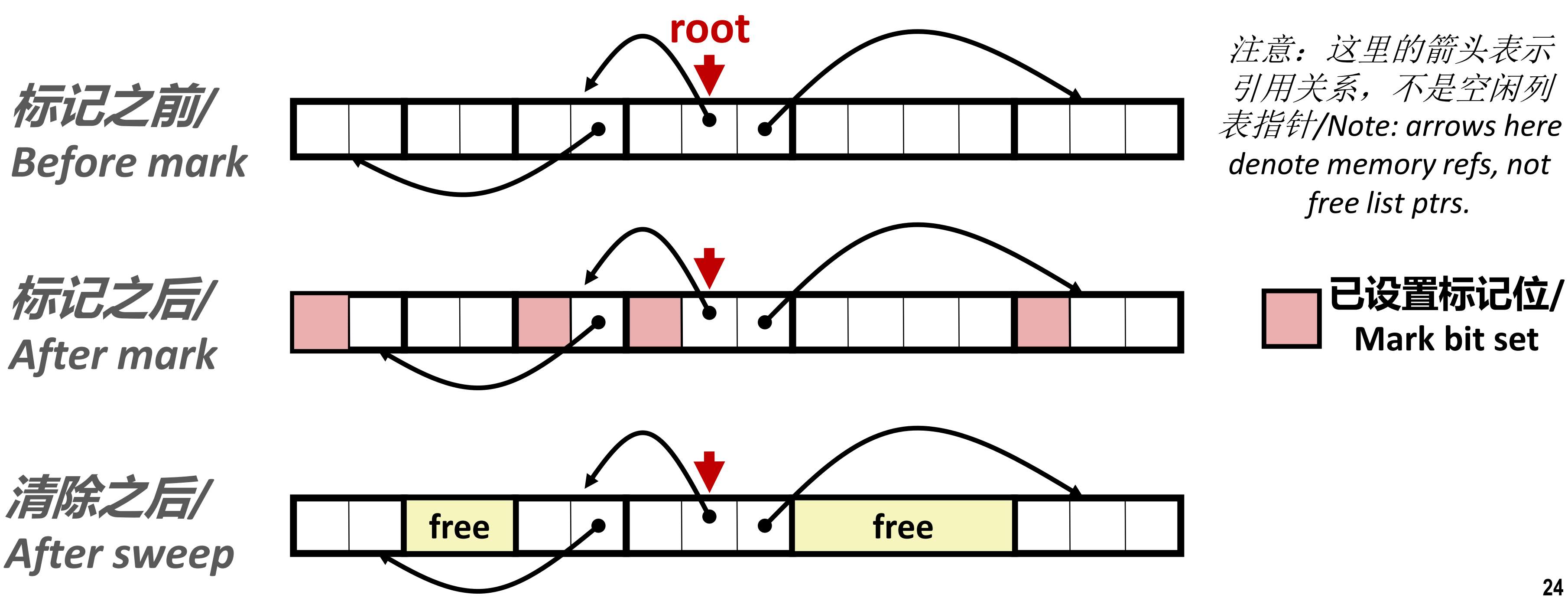


如果有从根节点到某个节点的路径则这个节点是可达的/A node (block) is *reachable* if there is a path from any root to that node.

不可达的都是垃圾（应用程序不再需要）/Non-reachable nodes are *garbage* (cannot be needed by the application)

标记清除收集算法/Mark and Sweep Collecting

- 可以基于malloc/free操作构建/**Can build on top of malloc/free package**
  - 一直使用malloc直到空间不够用/Allocate using `malloc` until you "run out of space"
- 什么时候内存不够用/**When out of space:**
  - 在每个块的头部使用额外的标记位/Use extra ***mark bit*** in the head of each block
  - ***Mark:*** 从根节点开始并对所有可达节点设置标记位/Start at roots and set mark bit on each reachable block
  - ***Sweep:*** 扫描所有的块并清除未标记的块/Scan all blocks and free blocks that are not marked



一个简单实现的前提假设/Assumptions For a Simple Implementation

- 应用/**Application**
  - **`new(n)`**：返回指向新块的指针，所有的域清除/returns pointer to new block with all locations cleared
  - **`read(b,i)`**：将块b中位置i的内容读到寄存器/read location **`i`** of block **`b`** into register
  - **`write(b,i,v)`**： 将v写入块b中的位置i/write **`v`** into location **`i`** of block **`b`**

- 每个块有一个头部字/**Each block will have a header word**
  - 对b可以使用b[-1]寻址/addressed as **`b[-1]`**, for a block **`b`**
  - 在不同的垃圾收集器里面有不同的用途/Used for different purposes in different collectors
- 垃圾收集器使用的操作/**Instructions used by the Garbage Collector**
  - **`is_ptr(p)`**：确定p是否是一个指针/determines whether **`p`** is a pointer
  - **`length(b)`**: 返回b的长度，不包括header/returns the length of block **`b`**, not including the header
  - **`get_roots()`**: 返回所有的根/returns all the roots

标记清除/Mark and Sweep (cont.)

通过内存图的深度优先遍历标记/Mark using depth-first traversal of the memory graph

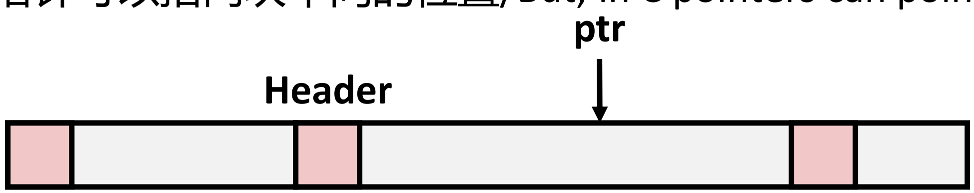
```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;         // do nothing if not pointer
   if (markBitSet(p)) return;      // check if already marked
   setMarkBit(p);                  // set the mark bit
   for (i=0; i < length(p); i++)   // call mark on all words
     mark(p[i]);                   //   in the block
   return;
}
```

清除阶段通过长度找到下一个块/Sweep using lengths to find next

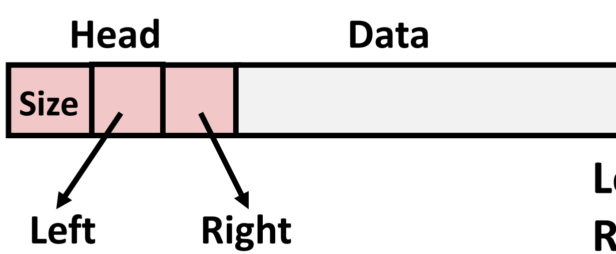
```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
}
```

C中保守的标记-清除算法/Conservative Mark & Sweep in C

- **C程序的一个保守垃圾收集器/A “conservative garbage collector” for C programs**
  - `is_ptr()` 用来判断一个字是否是指向一个已经分配的内存块的指针/**`is_ptr()`** determines if a word is a pointer by checking if it points to an allocated block of memory
  - 但是，C指针可以指向块中间的位置/But, in C pointers can point to the middle of a block



- 所以要如何找到块的开始？**/So how to find the beginning of the block?**
  - 可以使用一个平衡二叉树跟踪所有已经分配的块(key是块开始地址)/Can use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)
  - 平衡二叉树的指针可以存在head中(使用两个额外的字)/Balanced-tree pointers can be stored in header (use two additional words)



Left: smaller addresses/较小地址
Right: larger addresses/较大地址

内容提纲/Today

- 显示空闲列表/Explicit free lists
- 分离的空闲列表/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/Memory-Related Perils and Pitfalls

- 解引问题指针/Dereferencing bad pointers
- 使用未初始化内存/Reading uninitialized memory
- 覆盖内存/Overwriting memory
- 引用不存在的变量/Referencing nonexistent variables
- 重复释放内存块/Freeing blocks multiple times
- 引用释放的内存/Referencing freed blocks
- 释放内存失败/Failing to free blocks

C的操作符/C operators

Operators	*Associativity*
`() [] -> .`	left to right
`! ~ ++ -- + - * & (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= != <<= >>=`	right to left
`,`	left to right

- `->`, `()`, and `[]` have high precedence, with `*` and `&` just below
- Unary `+`, `-`, and `*` have higher precedence than binary forms

Source: K&R page 53

C指针申明：测试一下你自己/C Pointer Declarations: Test Yourself!

`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

解引问题指针/Dereferencing Bad Pointers

- 经典的scanf bug/The classic `scanf` bug

```
int val;

...

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

使用未初始化变量/Reading Uninitialized Memory

- 假设堆数据初始化为0/Assuming that heap data is initialized to zero

```
/* 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/Overwriting Memory

- 分配了可能错误大小的对象/Allocating the (possibly) wrong sized object

```
int **p;

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

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

覆盖内存/Overwriting Memory

- **Off-by-one error**

```
int **p;

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

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

覆盖内存/Overwriting Memory

- 没有检查最大字符串长度/Not checking the max string size

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

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

- 经典缓冲区溢出攻击的基础/Basis for classic buffer overflow attacks

覆盖内存/Overwriting Memory

- 指针运算理解错误/Misunderstanding pointer arithmetic

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

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

   return p;
}
```

覆盖内存/Overwriting Memory

- 引用了一个指针，而不是其指向的对象/Referencing a pointer instead of the object it points to

```
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/Referencing Nonexistent Variables

- 忘记函数返回之后局部变量不可用/Forgetting that local variables disappear when a function returns

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

   return &val;
}
```

多次重复释放块/Freeing Blocks Multiple Times

- **Nasty!**

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

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

引用已经释放的块/Referencing Freed Blocks

- **Evil!**

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

没有释放内存块（内存泄漏）/ Failing to Free Blocks (Memory Leaks)

- 慢性长期的问题/Slow, long-term killer!

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

没有释放内存块（内存泄漏）/Failing to Free Blocks (Memory Leaks)

- 只是释放了数据结构的一部分/Freeing only part of a data structure

```
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;
}
```

应对内存Bug/Dealing With Memory Bugs

- **调试器/Debugger: gdb**
  - 能够方便找出问题指针解引/Good for finding bad pointer dereferences
  - 难以探测其他内存问题/Hard to detect the other memory bugs
- **数据结构一致性检查/Data structure consistency checker**
  - 静默运行，出错时打印信息/Runs silently, prints message only on error
  - 用作错误归零的探针/Use as a probe to zero in on error
- **二进制翻译/Binary translator: `valgrind`**
  - 强大的调试和分析技术/Powerful debugging and analysis technique
  - 重写可执行目标文件的代码段/Rewrites text section of executable object file
  - 运行时检查每个单独的引用/Checks each individual reference at runtime
    - 问题指针、覆盖、越界访问/Bad pointers, overwrites, refs outside of allocated block
- **glibc malloc 包含了检查代码/glibc malloc contains checking code**
  - `setenv MALLOC_CHECK_ 3`