DYNAMIC MEMORY ALLOCATION: ADVANCED CONCEPTS

CS 045

Computer Organization and Architecture

Prof. Donald J. Patterson

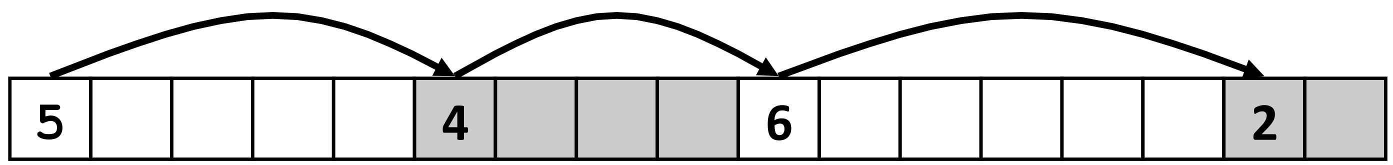
Adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

DYNAMIC MEMORY ALLOCATION: BASIC

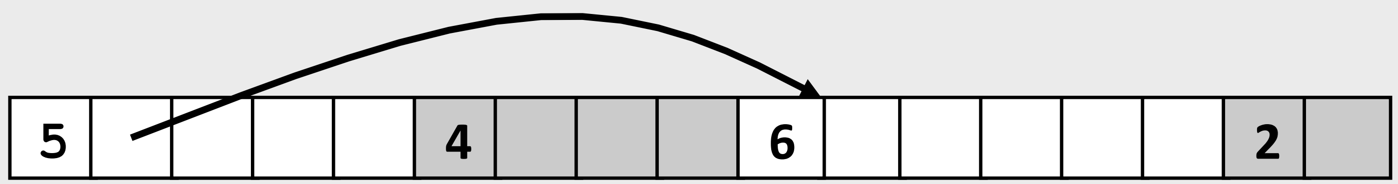
- EXPLICIT FREE LISTS
- SEGREGATED FREE LISTS
- GARBAGE COLLECTION
- MEMORY-RELATED PERILS AND PITFALLS

KEEPING TRACK OF FREE BLOCKS

- **Method 1:** ***Implicit free list*** **using length—links all blocks**



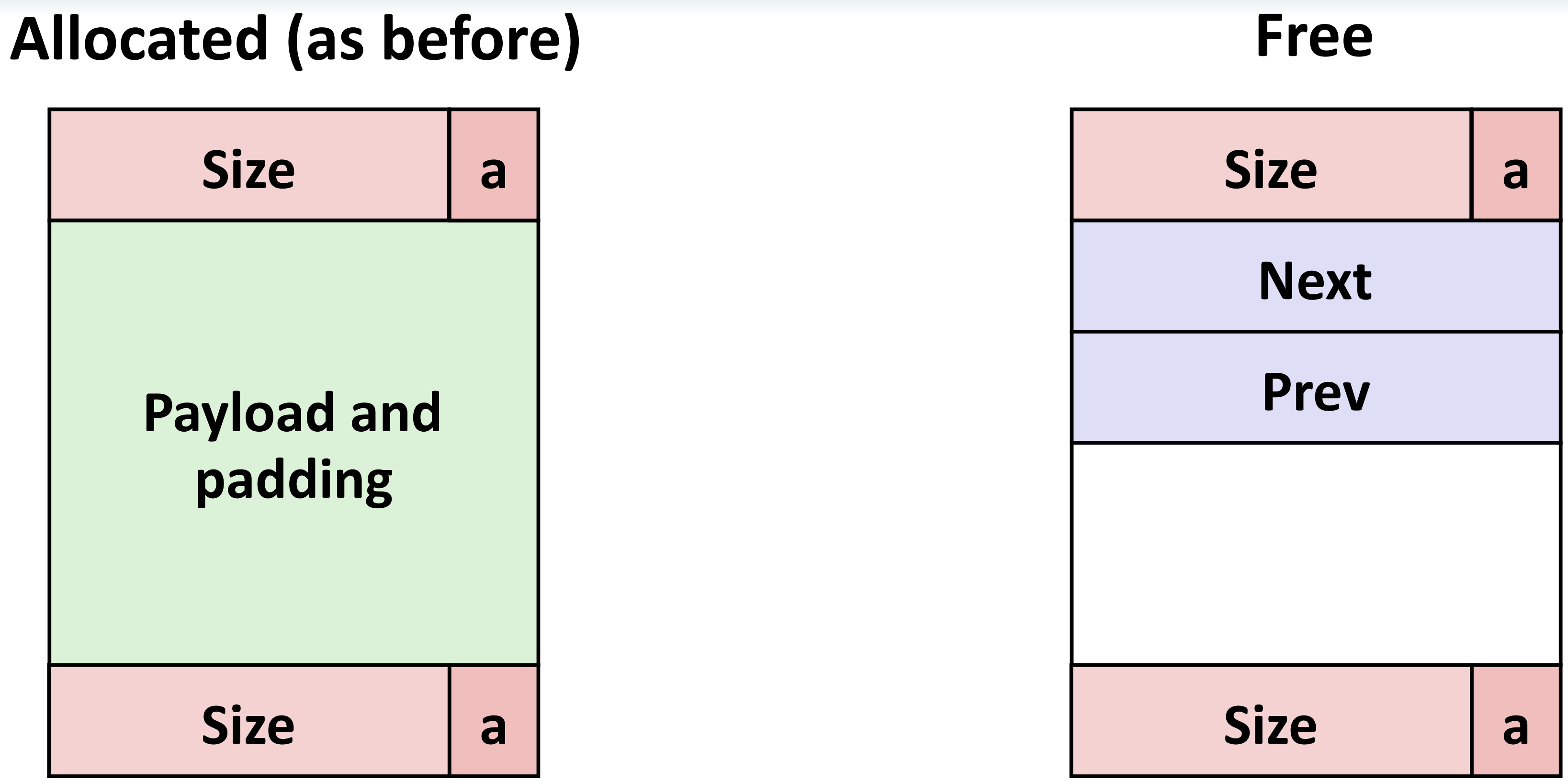
- **Method 2:** ***Explicit free list*** **among the free blocks using pointers**



- **Method 3:** ***Segregated free list***
  - Different free lists for different size classes

- **Method 4:** ***Blocks sorted by size***
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

EXPLICIT FREE LISTS

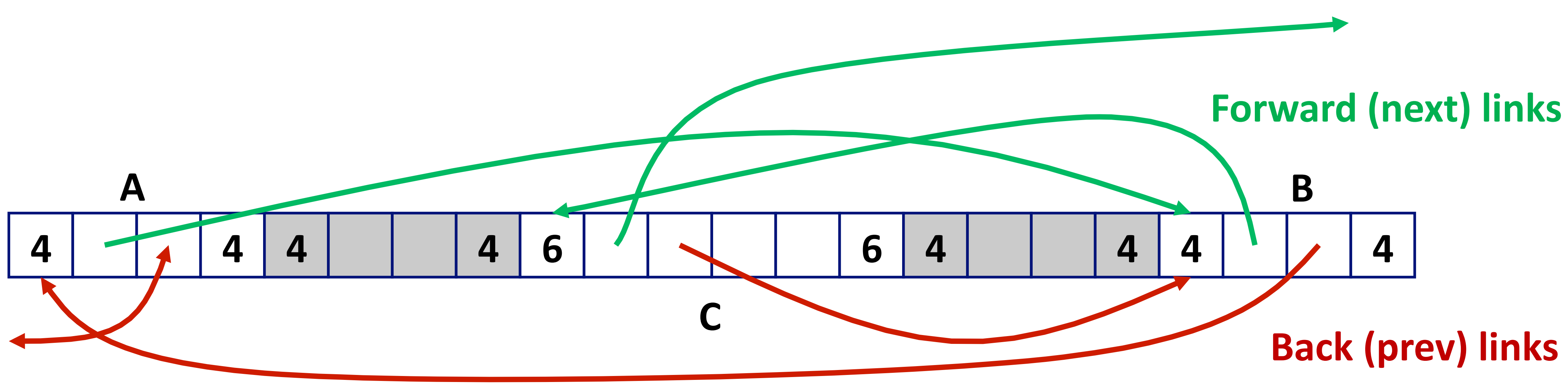


- **Maintain list(s) of *free* blocks, not *all* blocks**
  - The "next" free block could be anywhere
    - So we need to store forward/back pointers, not just sizes
  - Still need boundary tags for coalescing
  - Luckily we track only free blocks, so we can use payload area

EXPLICIT FREE LISTS

- **Logically:**

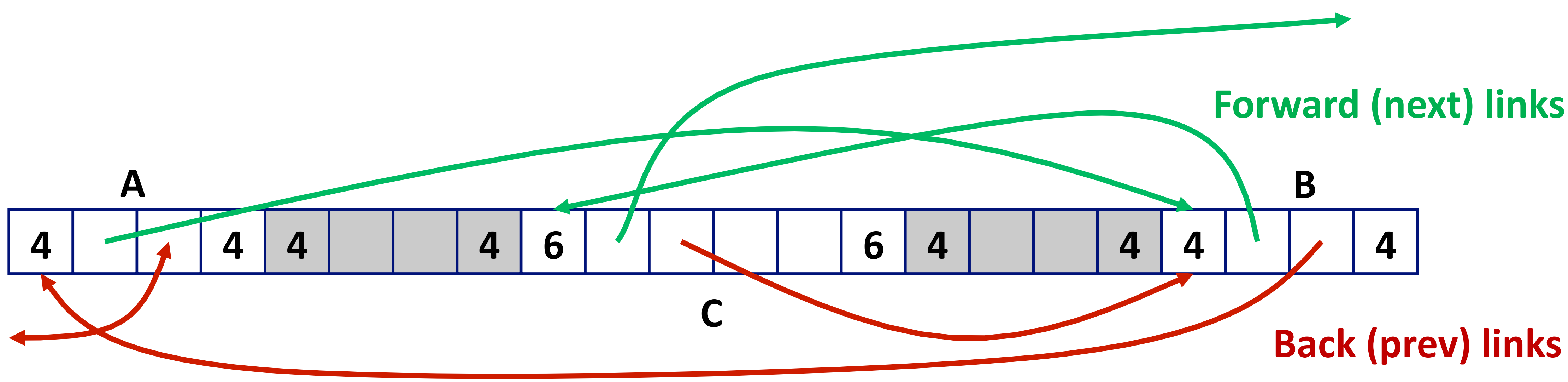
- **Physically: blocks can be in any order**



EXPLICIT FREE LISTS

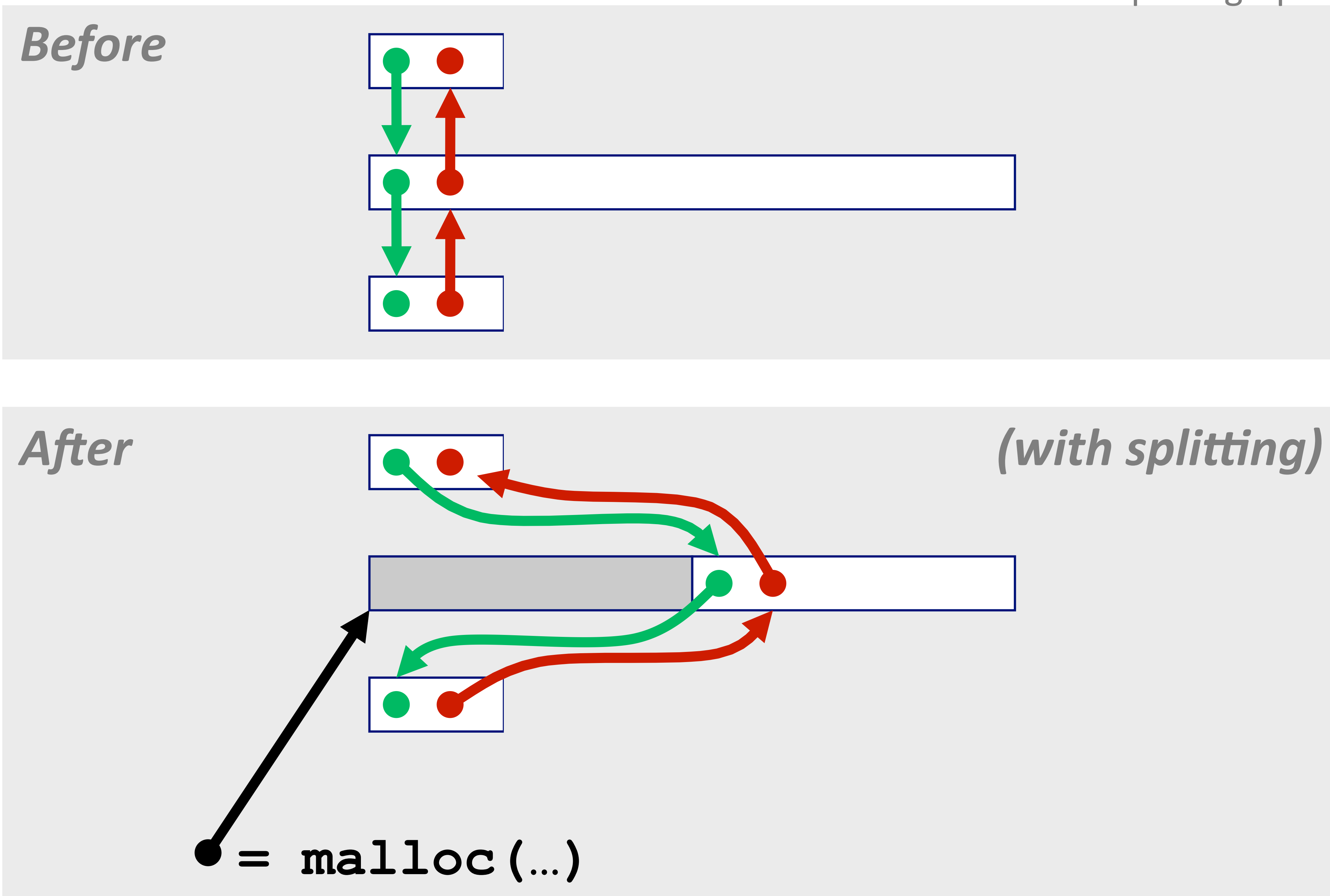
- **Logically:**

- **Physically: blocks can be in any order**



ALLOCATING FROM EXPLICIT FREE LISTS

conceptual graphic



FREEING WITH EXPLICIT FREE LISTS

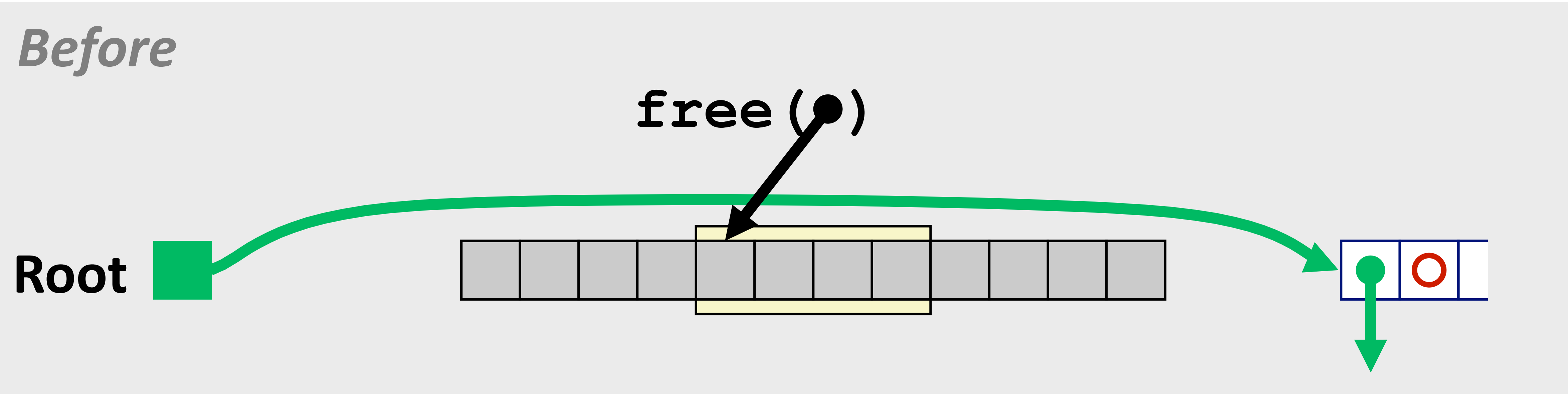
- ***Insertion policy*: Where in the free list do you put a newly freed block?**
- **LIFO (last-in-first-out) policy**
  - Insert freed block at the beginning of the free list
  - ***Pro:*** simple and constant time
  - ***Con:*** studies suggest fragmentation is worse than address ordered

- **Address-ordered policy**
  - Insert freed blocks so that free list blocks are always in address order:

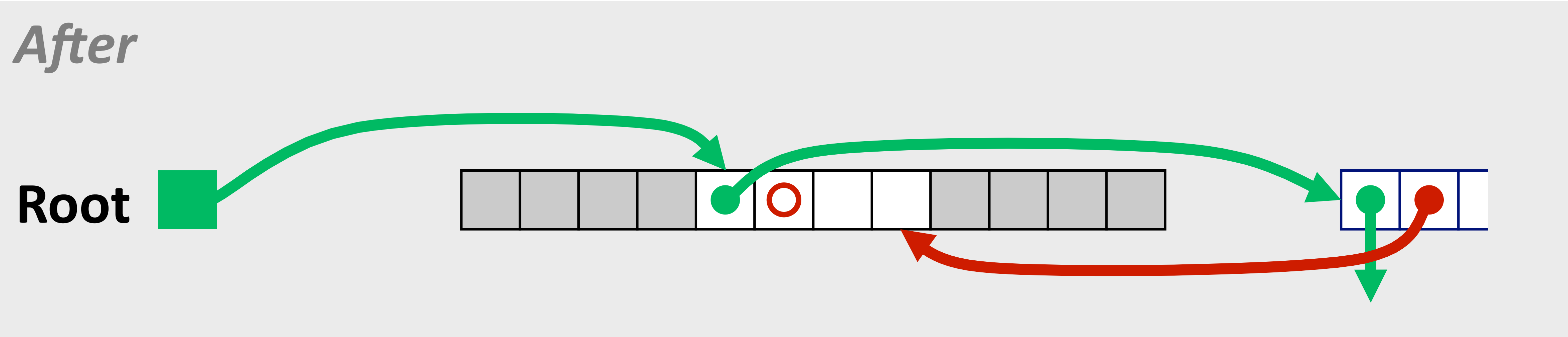
    addr(prev) < addr(curr) < addr(next)
  - ***Con:*** requires search
  - ***Pro:*** studies suggest fragmentation is lower than LIFO

FREEING WITH A LIFO POLICY (CASE 1)

conceptual graphic

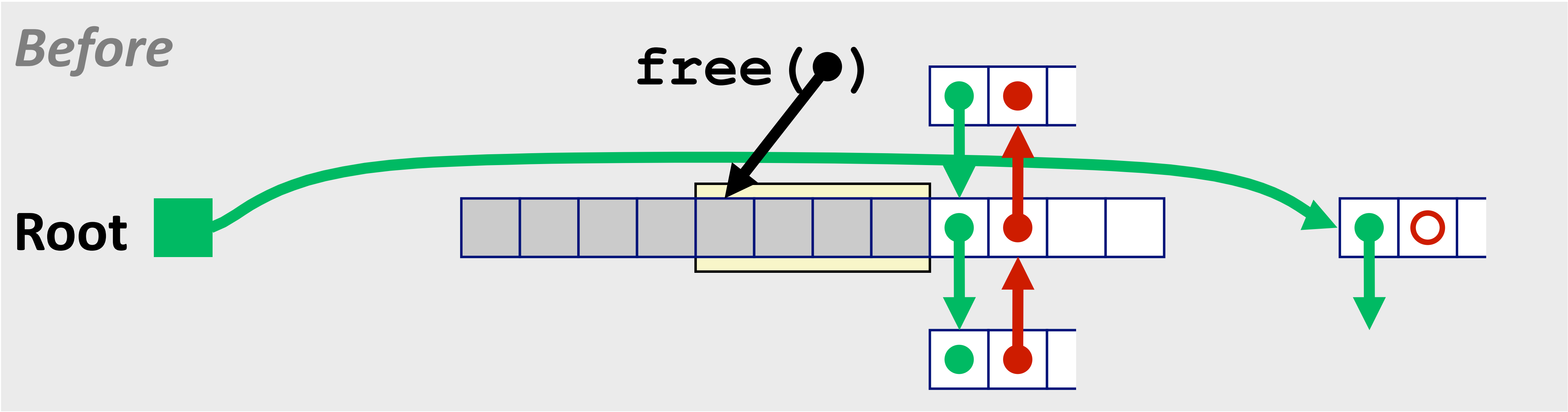


- Insert the freed block at the root of the list

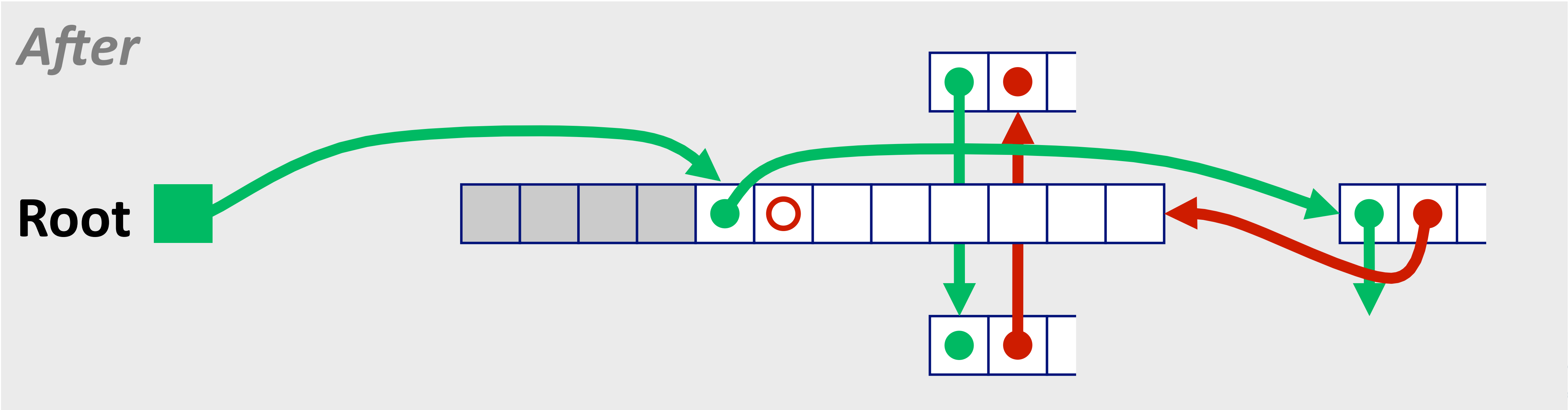


FREEING WITH A LIFO POLICY (CASE 2)

conceptual graphic

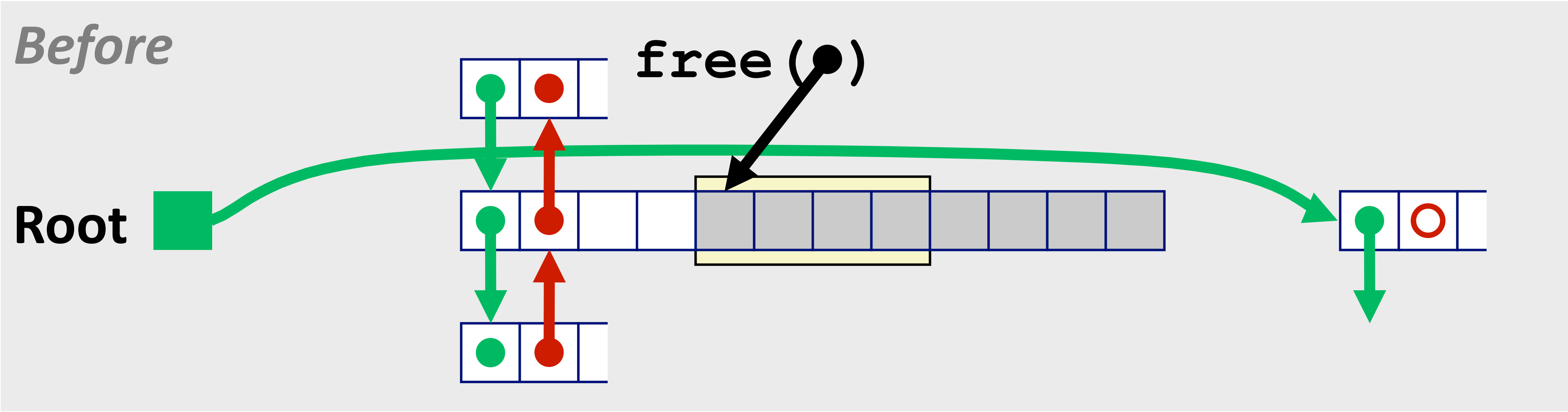


- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list

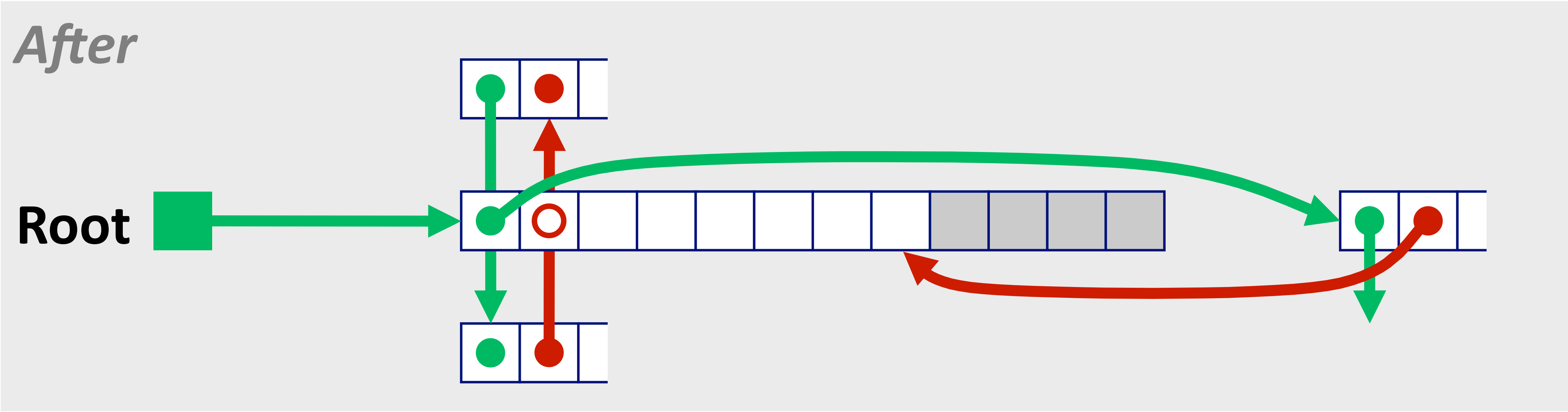


FREEING WITH A LIFO POLICY (CASE 3)

conceptual graphic

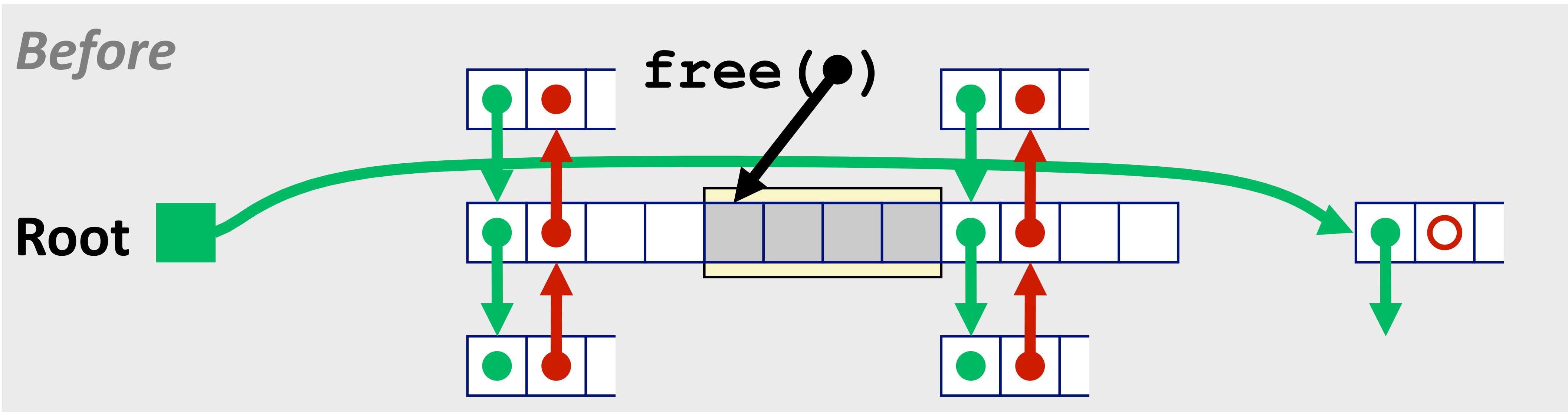


- **Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list**

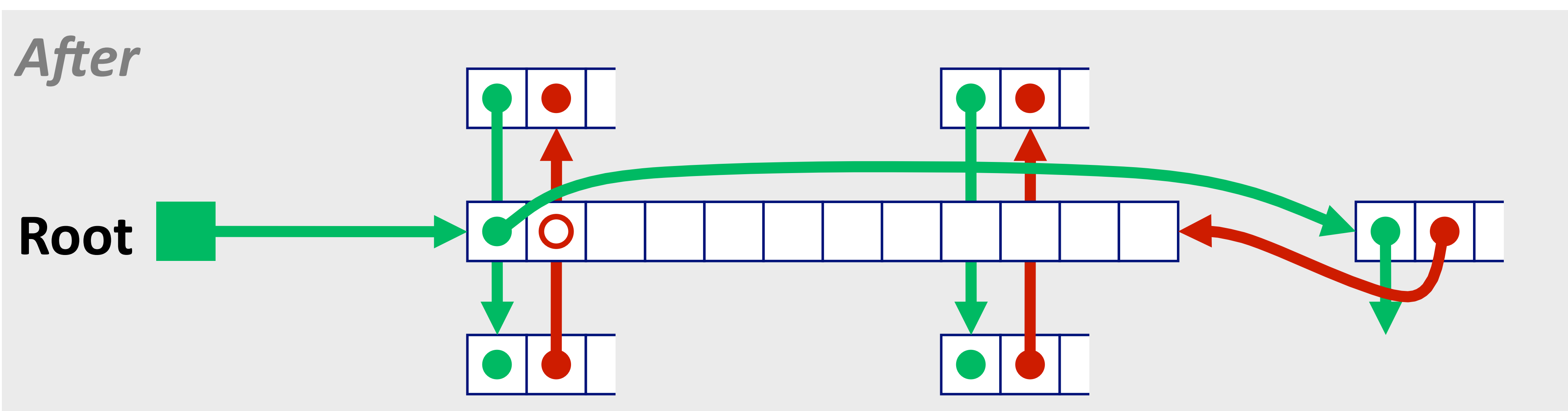


FREEING WITH A LIFO POLICY (CASE 4)

conceptual graphic



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



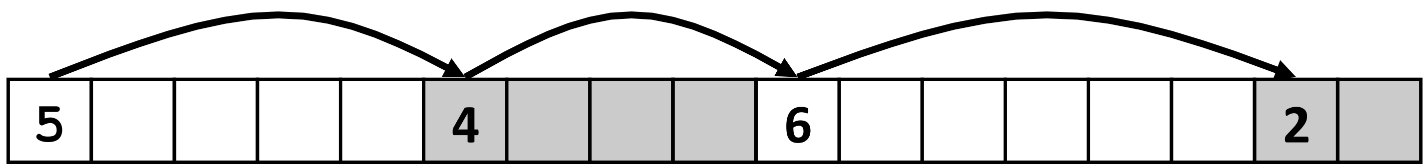
EXPLICIT LIST SUMMARY

- **Comparison to implicit list:**
  - Allocate is linear time in number of ***free*** blocks instead of ***all*** blocks
    - ***Much faster*** when most of the memory is full
  - Slightly more complicated allocate and free since needs to splice blocks in and out of the list
  - Some extra space for the links (2 extra words needed for each block)
    - Does this increase internal fragmentation?

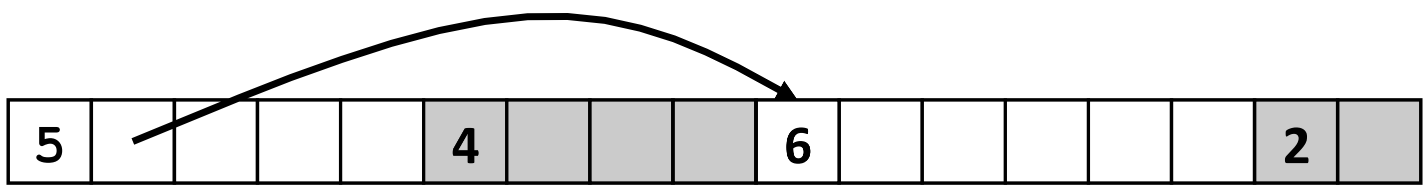
- **Most common use of linked lists is in conjunction with segregated free lists**
  - Keep multiple linked lists of different size classes, or possibly for different types of objects

KEEPING TRACK OF FREE BLOCKS

- **Method 1: *Implicit list* using length—links all blocks**



- **Method 2: *Explicit list* among the free blocks using pointers**



- **Method 3: *Segregated free list***
  - Different free lists for different size classes

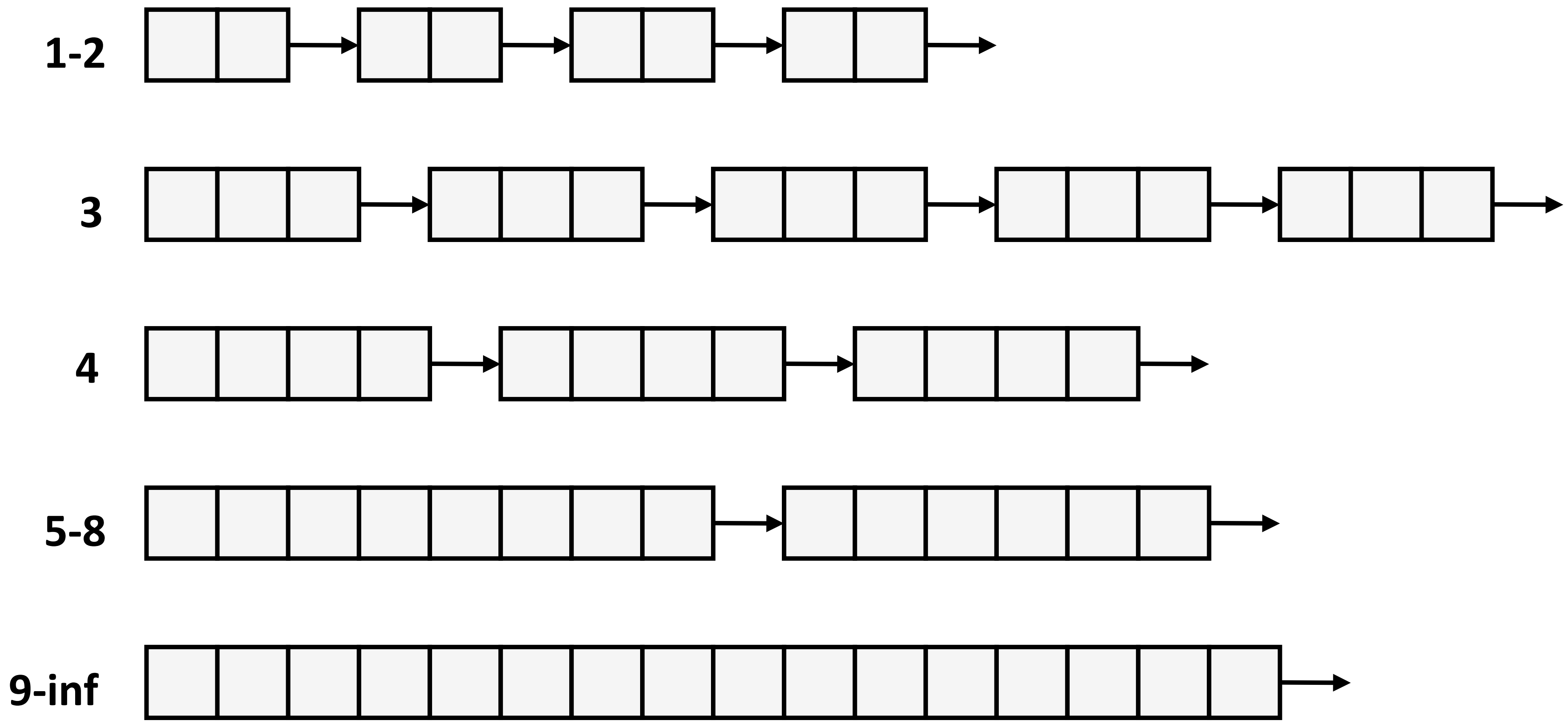
- **Method 4: *Blocks sorted by size***
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

DYNAMIC MEMORY ALLOCATION: BASIC

- EXPLICIT FREE LISTS
- SEGREGATED FREE LISTS
- GARBAGE COLLECTION
- MEMORY-RELATED PERILS AND PITFALLS

SEGREGATED LIST (SEGLIST) ALLOCATORS

- Each *size class* of blocks has its own free list



- Often have separate classes for each small size
- For larger sizes: One class for each two-power size

SEGLIST ALLOCATOR

- **Given an array of free lists, each one for some size class**

- **To allocate a block of size *n*:**
  - Search appropriate free list for block of size $m > n$
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list (optional)
  - If no block is found, try next larger class
  - Repeat until block is found

- **If no block is found:**
  - Request additional heap memory from OS (using **`sbrk()`**)
  - Allocate block of *n* bytes from this new memory
  - Place remainder as a single free block in largest size class.

SEGLIST ALLOCATOR (CONT.)

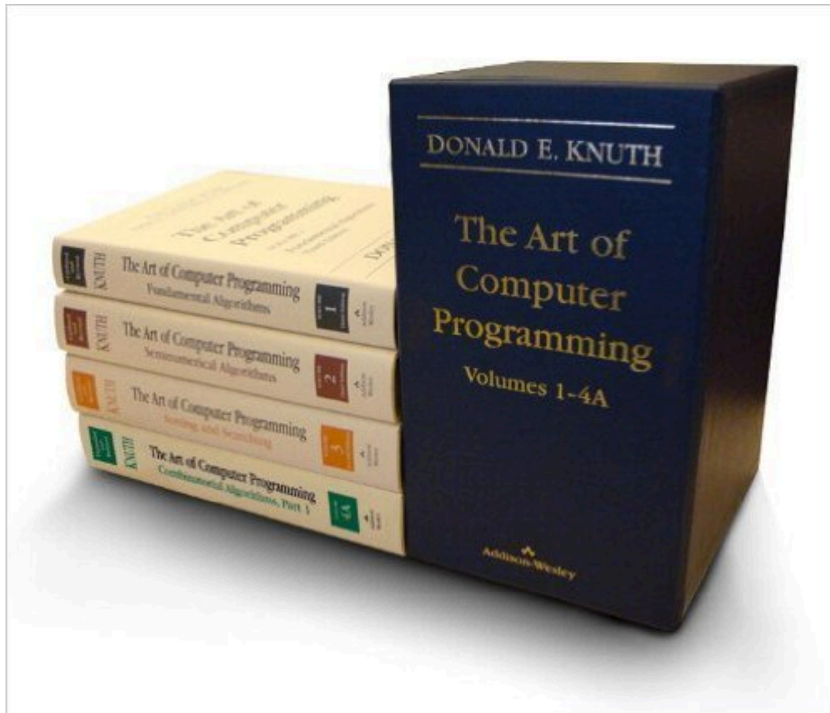
- **To free a block:**
  - Coalesce and place on appropriate list

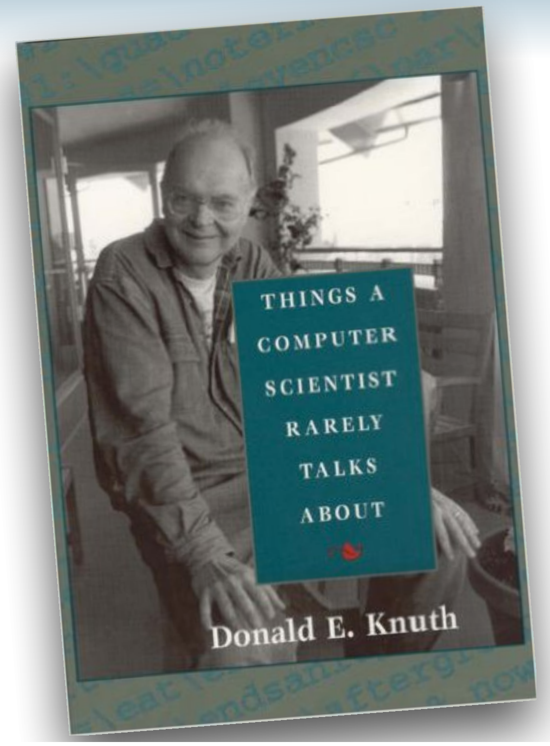
- **Advantages of seglist allocators**
  - Higher throughput
    - log time for power-of-two size classes
  - Better memory utilization
    - First-fit search of segregated free list approximates a best-fit search of entire heap.
    - Extreme case: Giving each block its own size class is equivalent to best-fit.

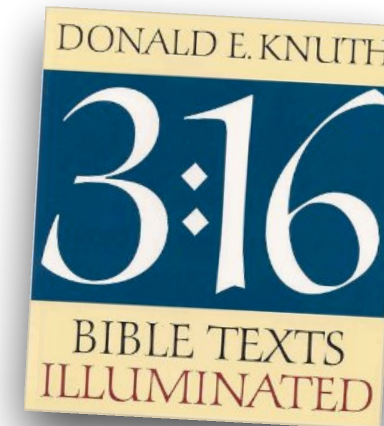
MORE INFO ON ALLOCATORS

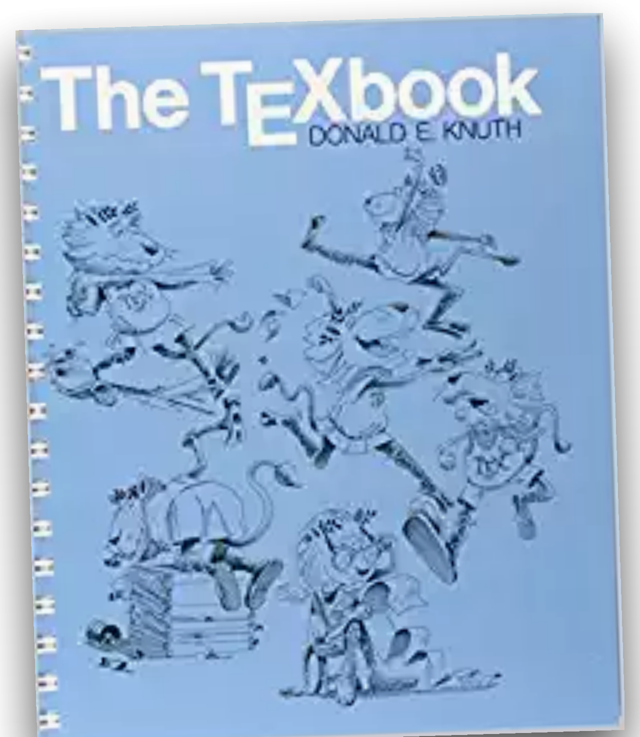
- D. Knuth, “The Art of Computer Programming”, 2nd edition, Addison Wesley, 1973
  - The classic reference on dynamic storage allocation
- Wilson et al, “Dynamic Storage Allocation: A Survey and Critical Review”, Proc. 1995 Int’l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.
  - Comprehensive survey
  - Available on Canvas

SIDENOTE: FAITH AND COMPUTER SCIENCE









How does a computer scientist understand infinity? What can probability theory teach us about free will? Can mathematical notions be used to enhance one's personal understanding of the Bible?

Perhaps no one is more qualified to address these questions than Donald E. Knuth, whose massive contributions to computing have led others to nickname him "The Father of Computer Science"—and whose religious faith led him to understand a fascinating analysis of the Bible called the 3:16 project. In this series of six spirited, informal lectures, Knuth explores the relationships between his vocation and his faith, revealing the unique perspective that his work with computing has lent to his understanding of God.

His starting point is the 3:16 project, an application of mathematical "random sampling" to the books of the Bible. The first lectures tell the story of the project's conception and execution, exploring its many dimensions of language translation, aesthetics, and theological history. Along the way, Knuth explains the many insights he gained from such interdisciplinary work. These theological musings culminate in a surprising final lecture tackling the ideas of infinity, free will, and some of the other big questions that lie at the juncture of theology and computation.

Things a Computer Scientist Rarely Talks About, with its charming and user-friendly format—each lecture ends with a question and answer exchange, and the book itself contains more than 100 illustrations—is a readable and intriguing approach to a crucial topic, certain to edify both those who are serious and curious about their faiths and those who look at the science of computation and wonder what it might teach them about their spiritual world.

from Publisher's comments on "Things"

DYNAMIC MEMORY ALLOCATION: BASIC

- EXPLICIT FREE LISTS
- SEGREGATED FREE LISTS
- GARBAGE COLLECTION
- MEMORY-RELATED PERILS AND PITFALLS

IMPLICIT MEMORY MANAGEMENT: GARBAGE COLLECTION

- ***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 */
}
```

- **Common in many dynamic languages:**
  - Python, Ruby, Java, Perl, ML, Lisp, Mathematica

- **Variants ("conservative" garbage collectors) exist for C and C++**
  - However, cannot necessarily collect all garbage

GARBAGE COLLECTION

- **How does the memory manager know when memory can be freed?**
  - In general we cannot know what is going to be used in the future since it depends on conditionals
  - But we can tell that certain blocks cannot be used if there are no pointers to them

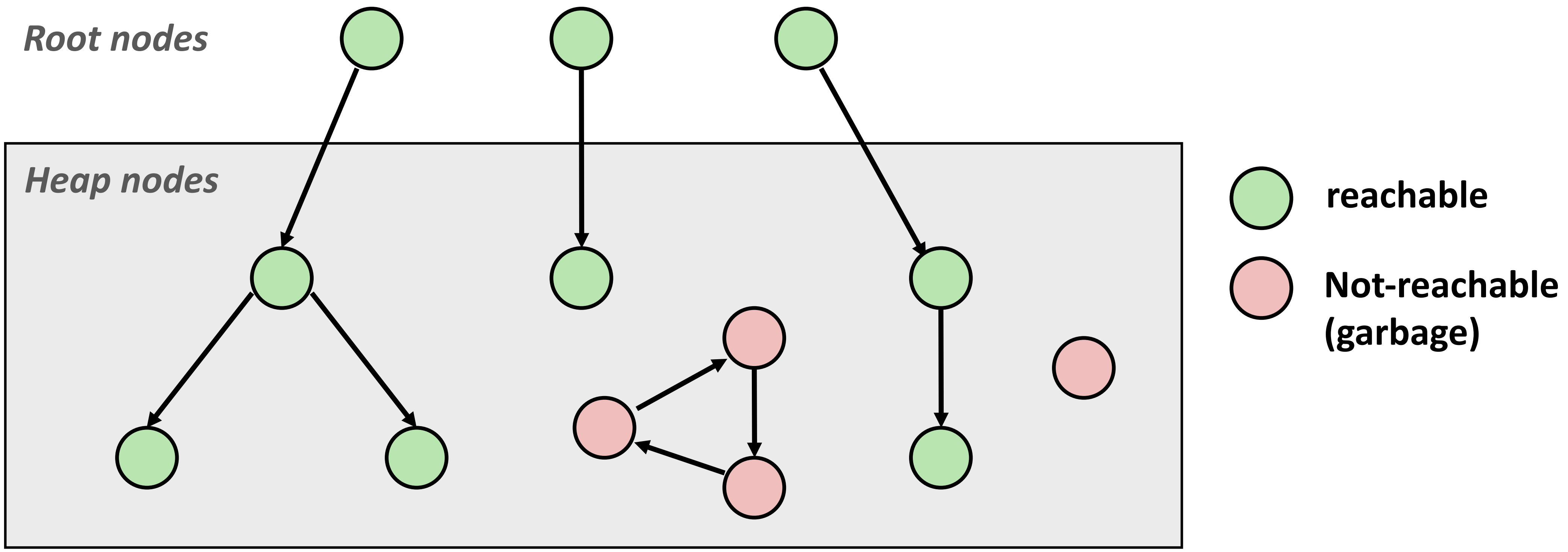
- **Must make certain assumptions about pointers**
  - Memory manager can distinguish pointers from non-pointers
  - All pointers point to the start of a block
  - Cannot hide pointers
    (e.g., by coercing them to an `int`, and then back again)

CLASSIC GC ALGORITHMS

- **Mark-and-sweep collection (McCarthy, 1960)**
  - Does not move blocks (unless you also "compact")
- **Reference counting (Collins, 1960)**
  - Does not move blocks (not discussed)
- **Copying collection (Minsky, 1963)**
  - Moves blocks (not discussed)
- **Generational Collectors (Lieberman and Hewitt, 1983)**
  - Collection based on lifetimes
    - Most allocations become garbage very soon
    - 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.

MEMORY AS A GRAPH

- **We view memory as a directed graph**
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called ***root*** nodes (e.g. registers, locations on the stack, global variables)

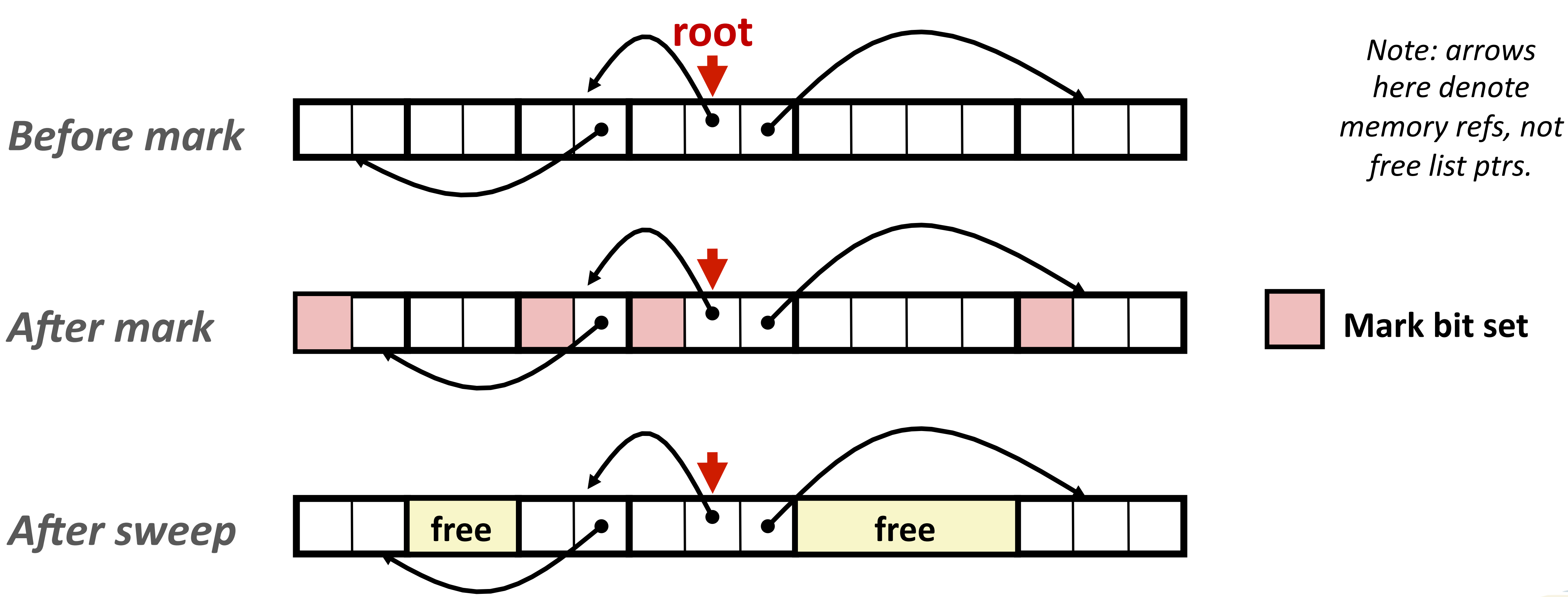


A node (block) is *reachable* if there is a path from any root to that node.

Non-reachable nodes are *garbage* (cannot be needed by the application)

MARK AND SWEEP COLLECTING

- **Can build on top of malloc/free package**
  - Allocate using `malloc` until you "run out of space"
- **When out of space:**
  - Use extra ***mark bit*** in the head of each block
  - ***Mark:*** Start at roots and set mark bit on each reachable block
  - ***Sweep:*** Scan all blocks and free blocks that are not marked



ASSUMPTIONS FOR A SIMPLE IMPLEMENTATION

- **Application**
  - **`new(n)`**: returns pointer to new block with all locations cleared
  - **`read(b,i)`**: read location **`i`** of block **`b`** into register
  - **`write(b,i,v)`**: write **`v`** into location **`i`** of block **`b`**

- **Each block will have a header word**
  - addressed as **`b[-1]`**, for a block **`b`**
  - Used for different purposes in different collectors

- **Instructions used by the Garbage Collector**
  - **`is_ptr(p)`**: determines whether **`p`** is a pointer
  - **`length(b)`**: returns the length of block **`b`**, not including the header
  - **`get_roots()`**: returns all the roots

MARK AND SWEEP (CONT.)

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;
}
```

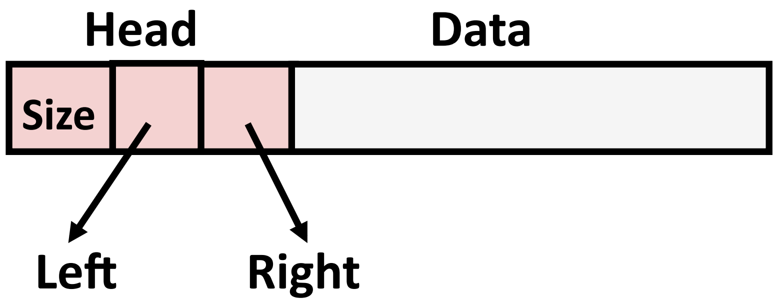
Sweep using lengths to find next block

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

CONSERVATIVE MARK & SWEEP IN C

- **A "conservative garbage collector" for C programs**
  - `is_ptr()` determines if a word is a pointer by checking if it points to an allocated block of memory
  - But, in C pointers can point to the middle of a block

- **So how to find the beginning of the block?**
  - Can use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)
  - Balanced-tree pointers can be stored in header (use two additional words)



Left: smaller addresses
Right: larger addresses

DYNAMIC MEMORY ALLOCATION: BASIC

- EXPLICIT FREE LISTS
- SEGREGATED FREE LISTS
- GARBAGE COLLECTION
- MEMORY-RELATED PERILS AND PITFALLS

MEMORY-RELATED PERILS AND PITFALLS

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

C OPERATORS

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

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

Source: K&R page 53

DEREFERENCING BAD POINTERS

- **The classic `scanf` bug**

```
int val;

...

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

READING UNINITIALIZED MEMORY

- **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;
}
```

OVERWRITING MEMORY

- **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));
}
```

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));
}
```

OVERWRITING MEMORY

- **Not checking the max string size**

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

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

- **Basis for classic buffer overflow attacks**

OVERWRITING MEMORY

- **Misunderstanding pointer arithmetic**

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

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

   return p;
}
```

OVERWRITING MEMORY

- 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);
}
```

REFERENCING NONEXISTENT VARIABLES

- **Forgetting that local variables disappear when a function returns**

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

   return &val;
}
```

FREEING BLOCKS MULTIPLE TIMES

- **Nasty!**

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

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

DEALING WITH MEMORY BUGS

- **Debugger: `gdb`**
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs
- **Data structure consistency checker**
  - Runs silently, prints message only on error
  - Use as a probe to zero in on error
- **Binary translator: `valgrind`**
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Checks each individual reference at runtime
    - Bad pointers, overwrites, refs outside of allocated block
- **glibc malloc contains checking code**
  - `setenv MALLOC_CHECK_ 3`

WESTMONT INSPIRED

COMPUTING LAB

SUMMARY