Dynamic Memory Allocation: Advanced Concepts

System Programming

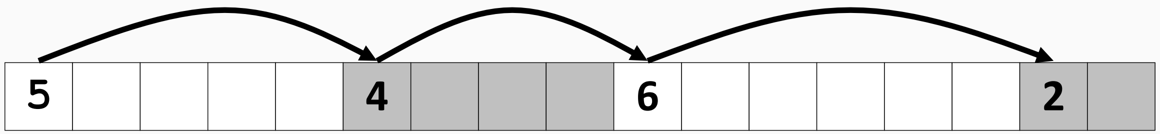
Woong Sul

Today

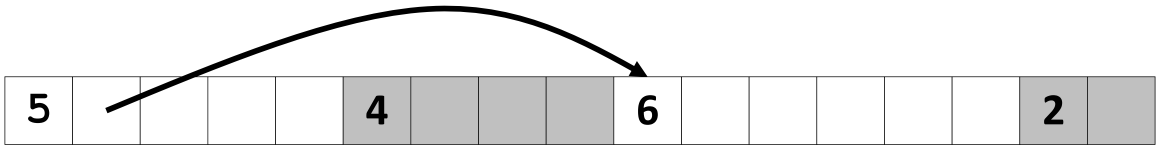
- Explicit free lists
- Segregated free lists
- Garbage collection
- Memory-related perils and pitfalls

Keeping Track of Free Blocks

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



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



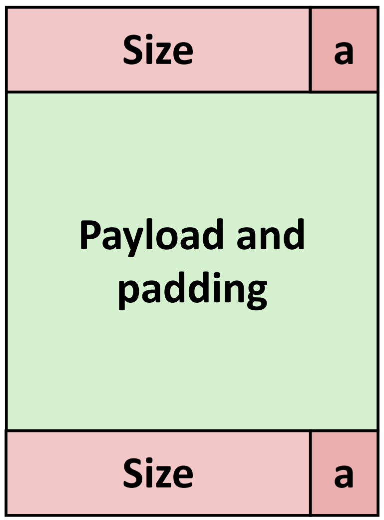
- Method 3: *Segregated free list*
  - Different free lists for different size classes

- 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

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

Allocated (as before)

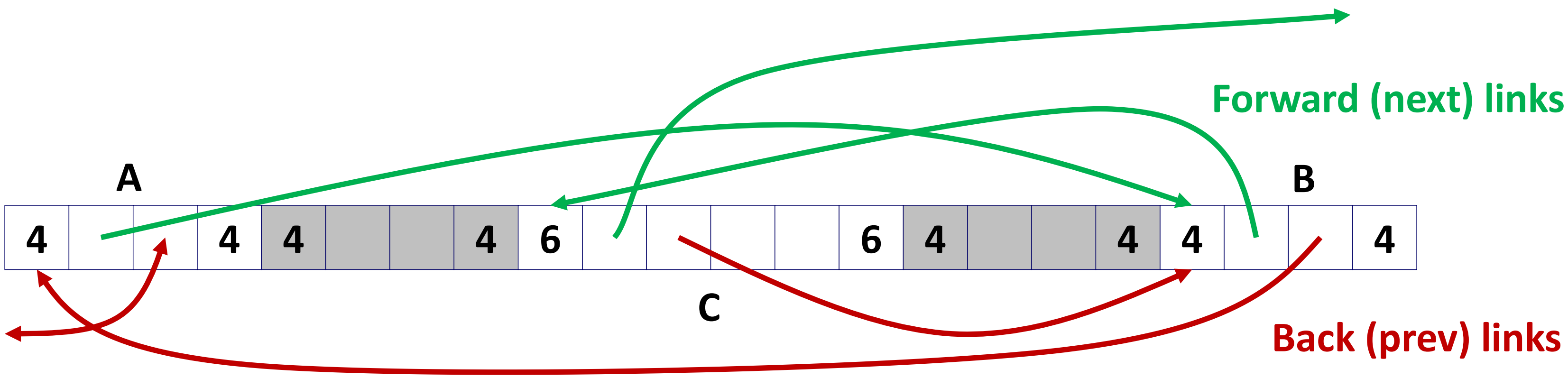


Free

Explicit Free Lists

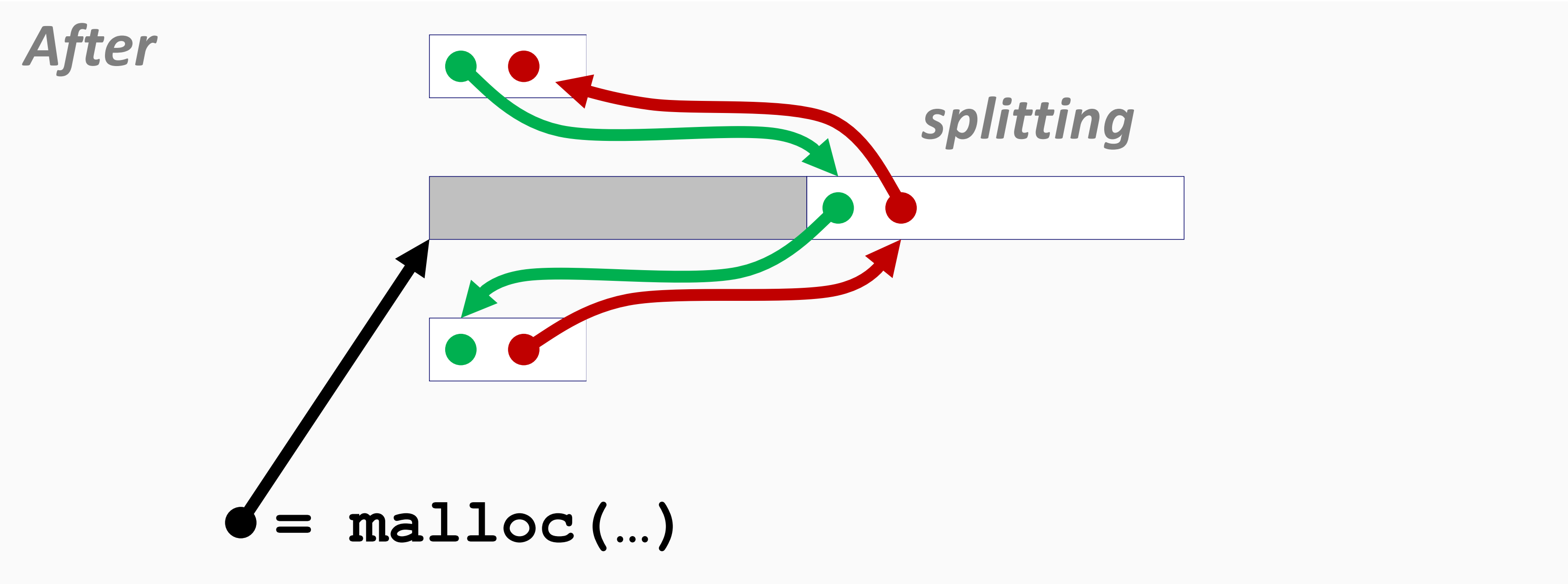
- Logically:

- 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:

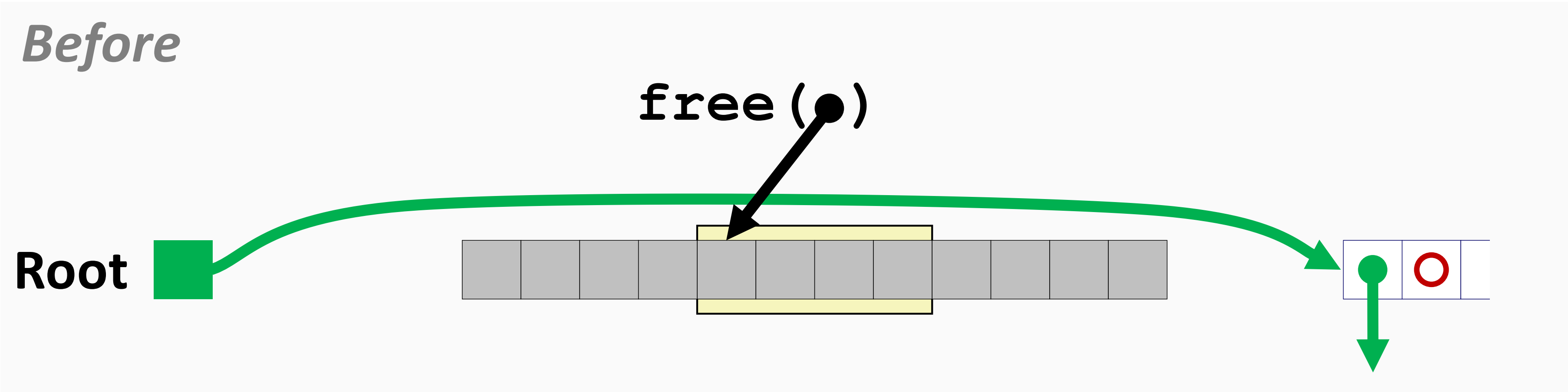
    addr(prev) < addr(curr) < addr(next)

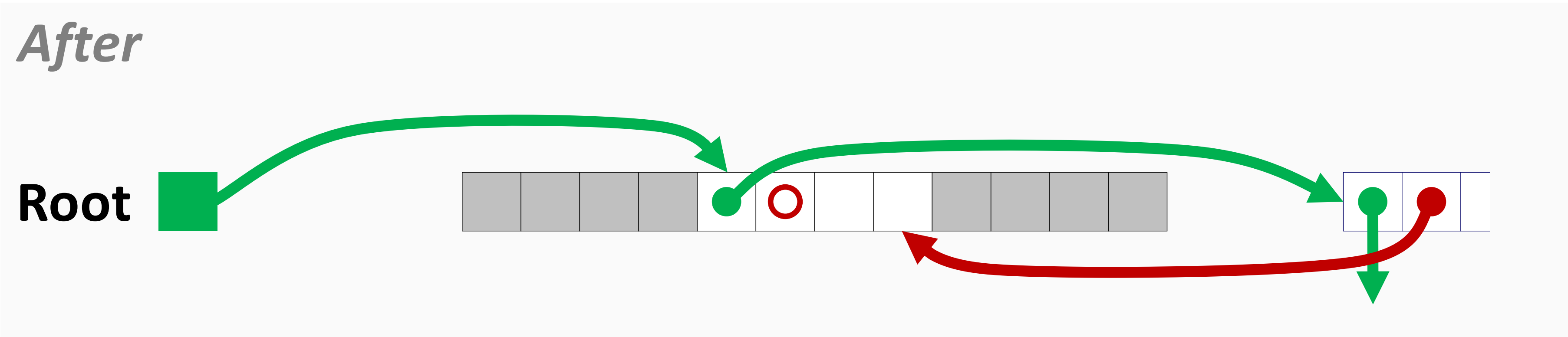
  - ***Con***: requires search
  - ***Pro***: studies suggest fragmentation is lower than LIFO

Freeing With a LIFO Policy (Case 1)

- Insert the freed block at the root of the list

conceptual graphic

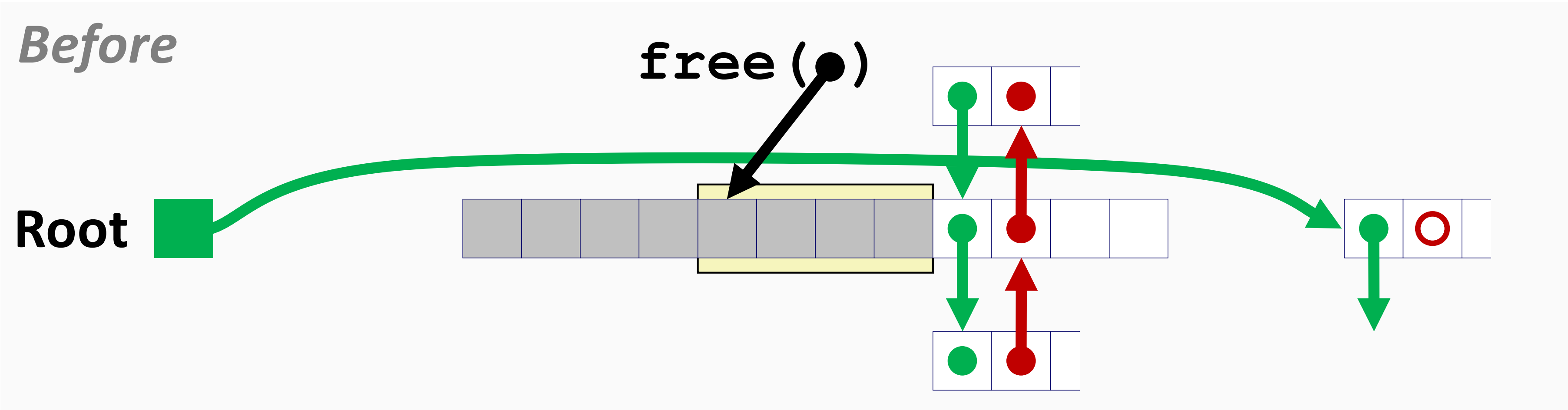


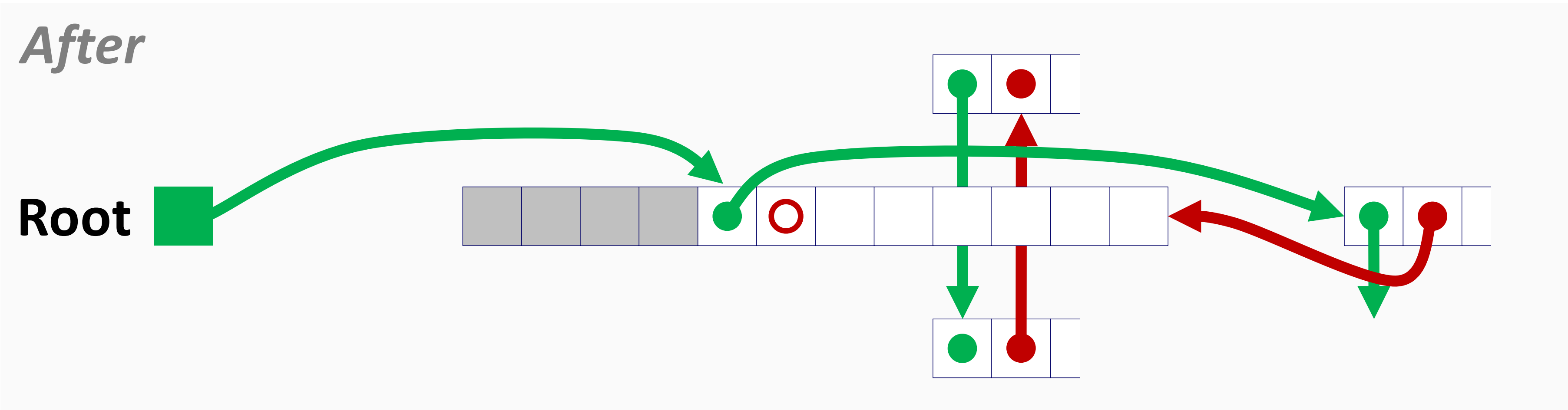


Freeing With a LIFO Policy (Case 2)

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

conceptual graphic

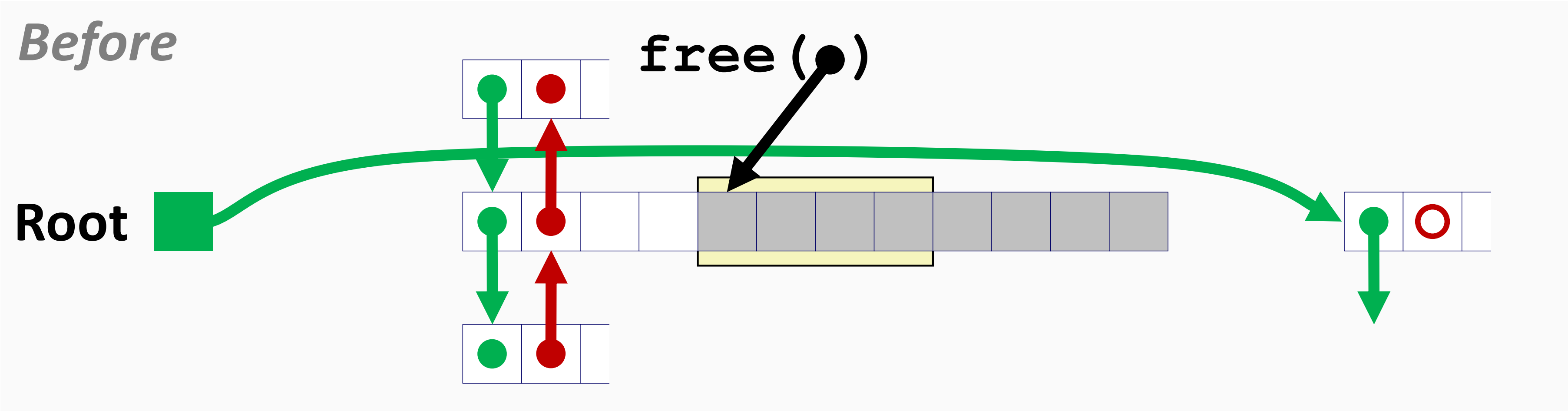


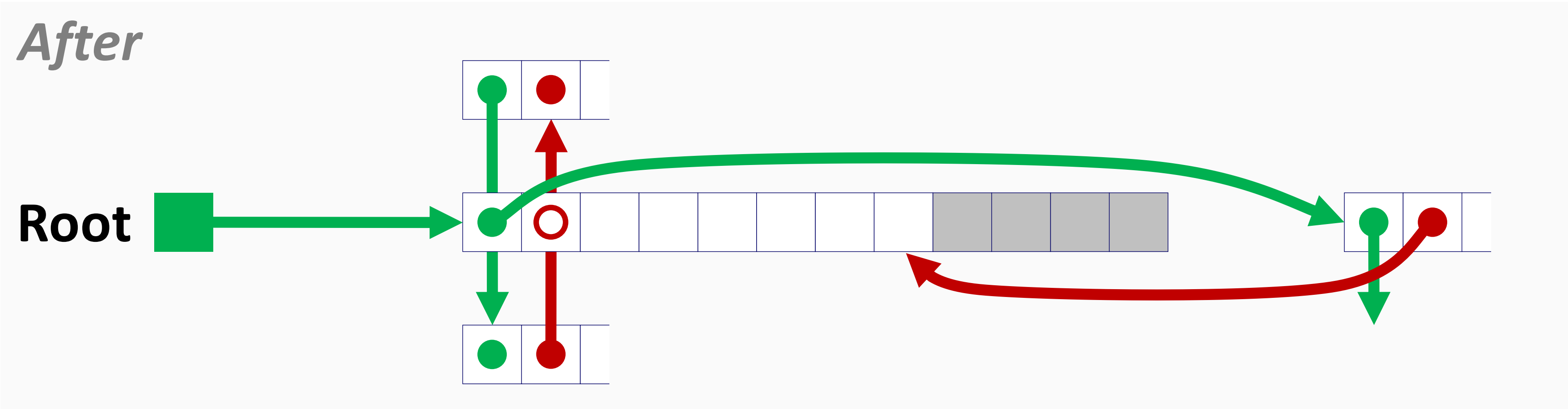


Freeing With a LIFO Policy (Case 3)

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

conceptual graphic

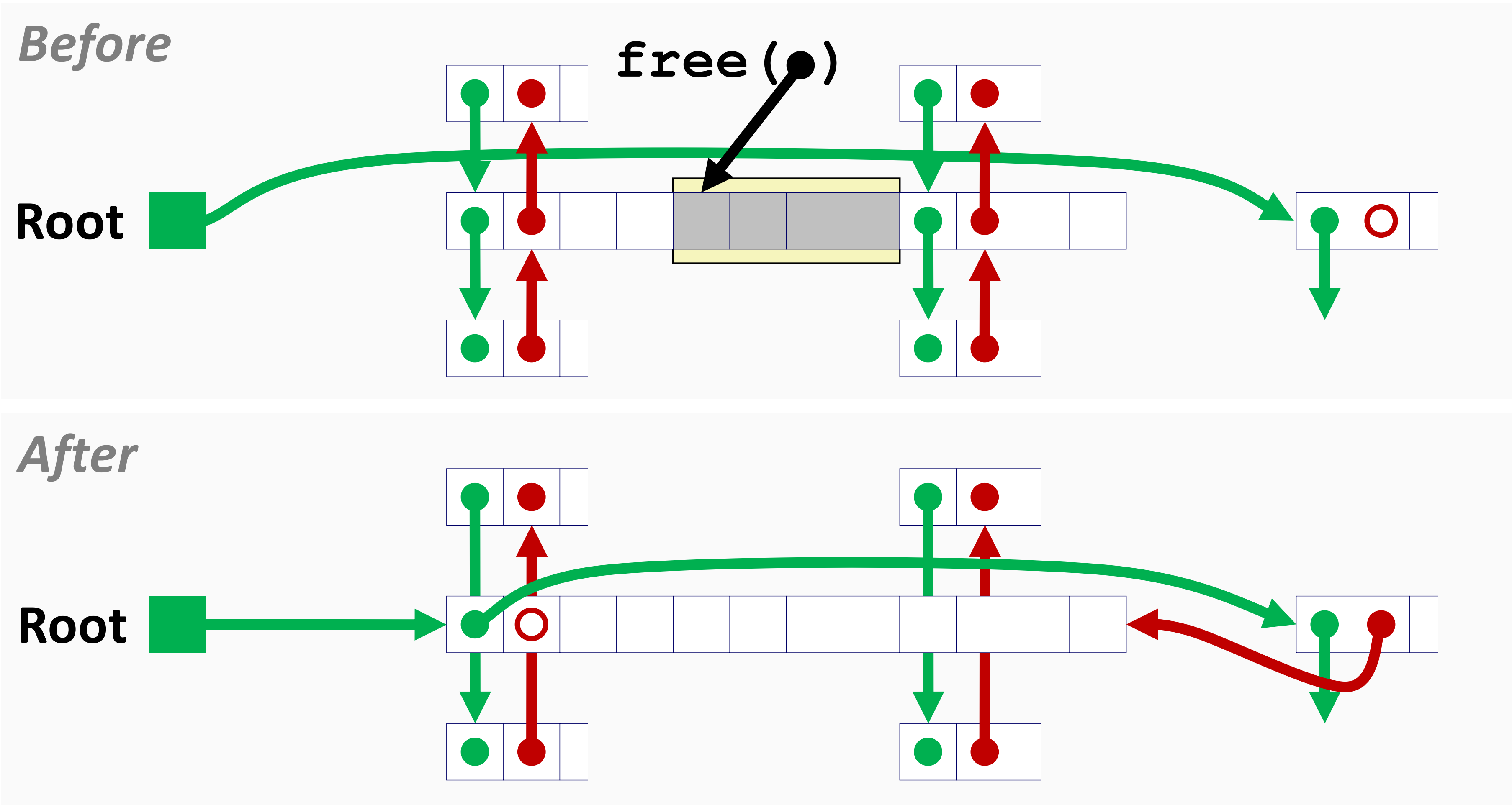




Freeing With a LIFO Policy (Case 4)

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

conceptual graphic



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?

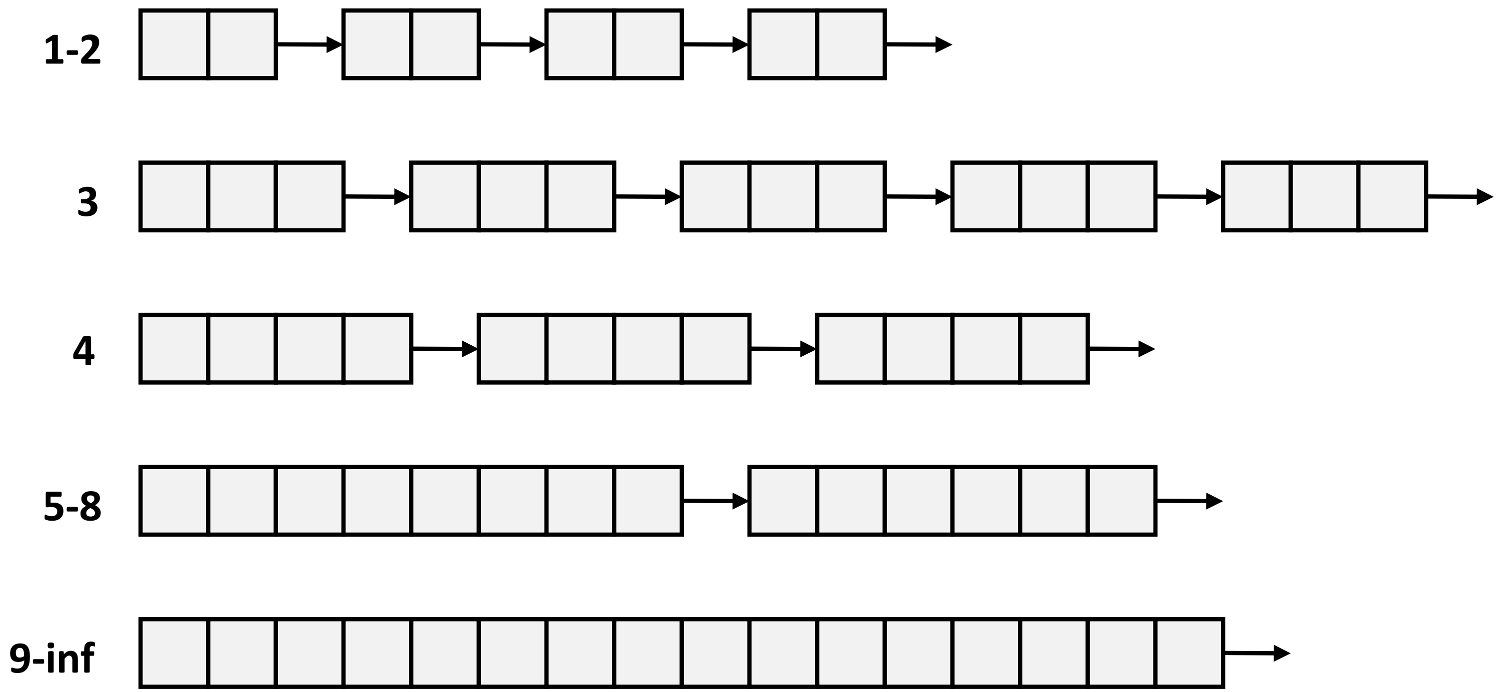
- Most common use of linked lists is in conjunction with segregated free lists
  - Keep multiple linked lists of different size classes, or possibly for different types of objects

Today

- 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 (Cnt'd)

- 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

Today

- Explicit free lists
- Segregated free lists
- Garbage collection
- Memory-related perils and pitfalls

Implicit Memory Management

- ***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/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

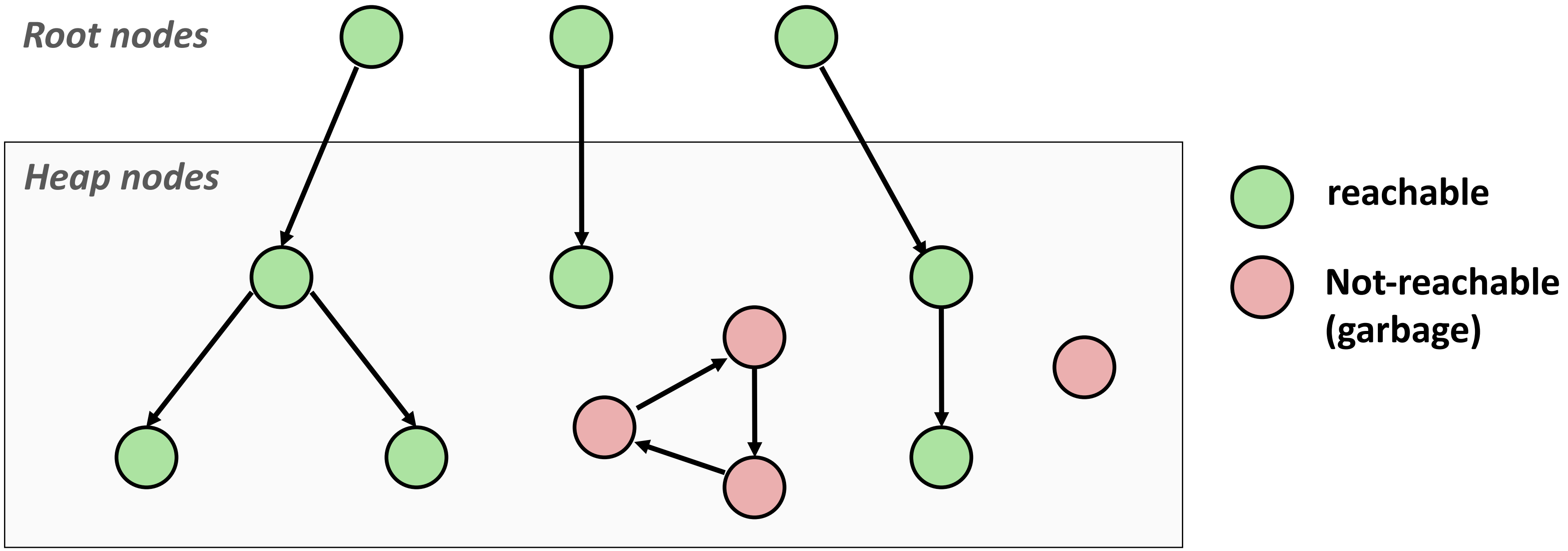
- Must make certain ***assumptions*** about pointers
  - Memory manager can distinguish pointers from non-pointers
  - All pointers point to the start of a block
  - Cannot hide pointers
    (e.g., by coercing them to an **`int`**, and then back again)

Classical 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

- Each block/pointer is a node/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)

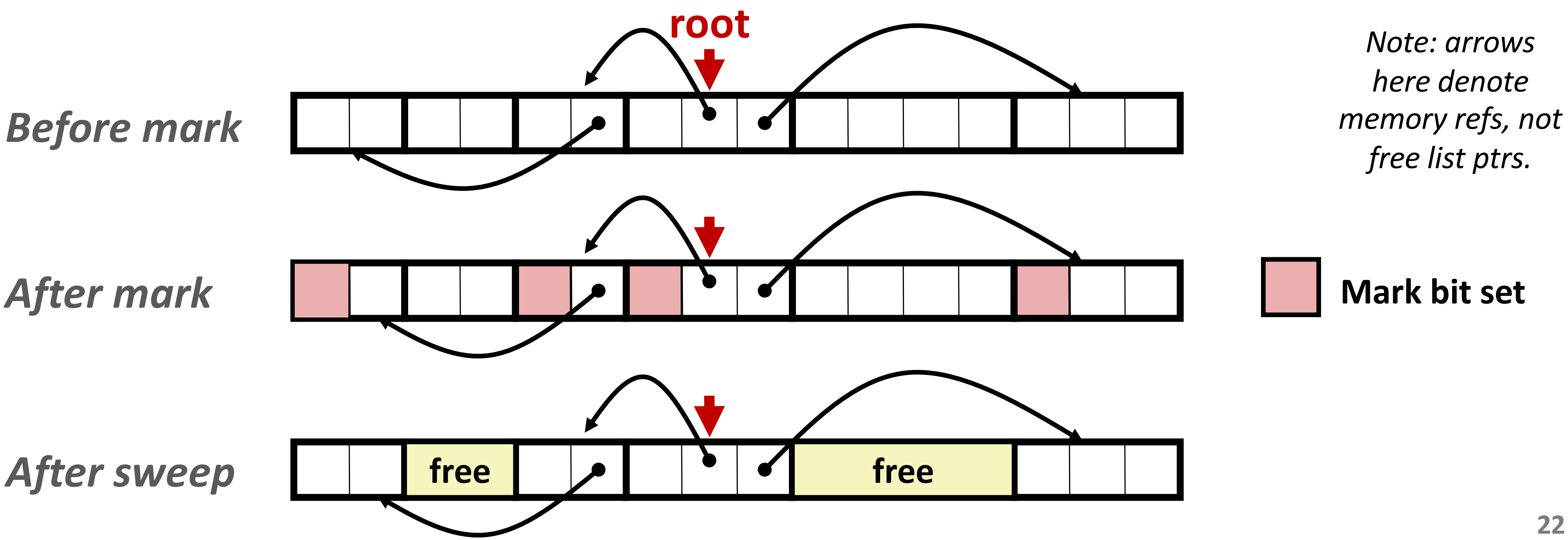


A node (block) is *reachable* if there is a path from any root to that node.

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 (Cnt'd)

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);
}
```

Today

- 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

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

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 – 1/5

- Allocating the (possibly) wrong sized object

```
int** p;

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

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

Overwriting Memory – 2/5

- 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 – 3/5

- 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 – 4/5

- Misunderstanding pointer arithmetic

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

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

   return p;
}
```

Overwriting Memory – 5/5

- 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 Non-existent 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);
```

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]++;
```

Failing to Free Blocks (Memory Leaks)

- Slow, long-term killer!

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

Failing to Free Blocks (Memory Leaks)

- 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;
}
```

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 (or assertion)
  - 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
```