CS61: Systems Programming and Machine Organization
Harvard University, Fall 2009

Lecture 11:

Dynamic Memory Allocation 2

Prof. Matt Welsh

October 8, 2009

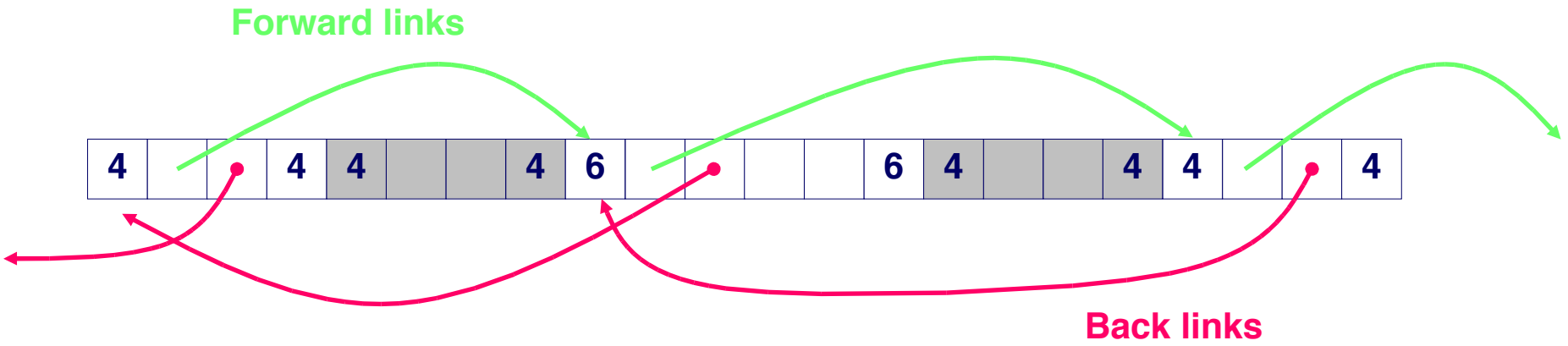
Topics for today

- Continuing discussion of dynamic memory allocation
- Explicit free list management
- Segregated free lists
- Implicit memory management: Garbage collection
- Common memory bugs

Explicit Free Lists

Explicit pointers in block header to next and previous free blocks

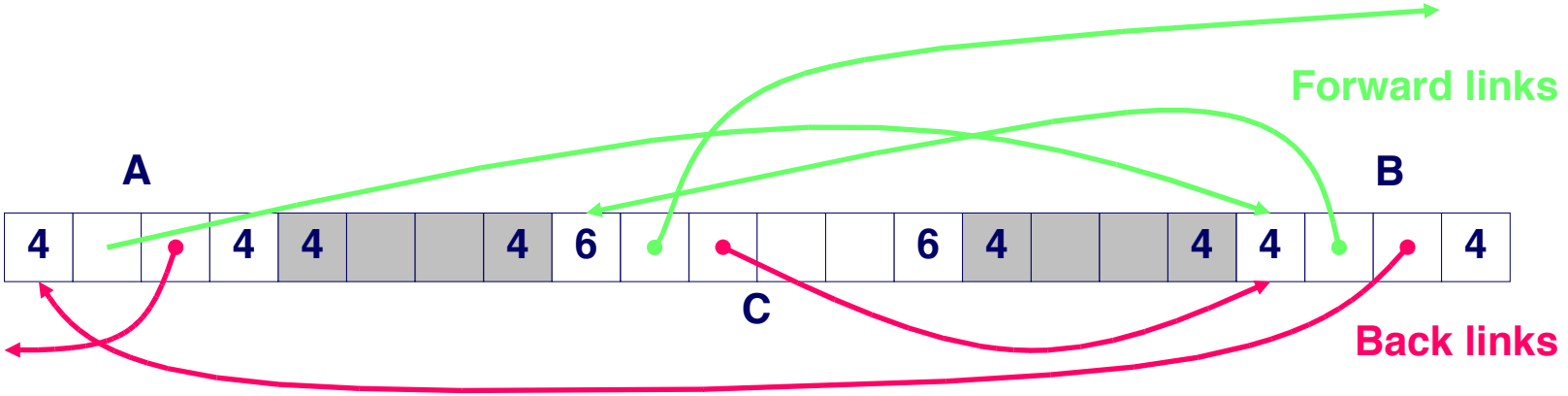
- Just a doubly-linked list.
- No pointers to or from *allocated* blocks: *Can put the pointers into the payload!*
- Still need boundary tags, in order to perform free block coalescing.



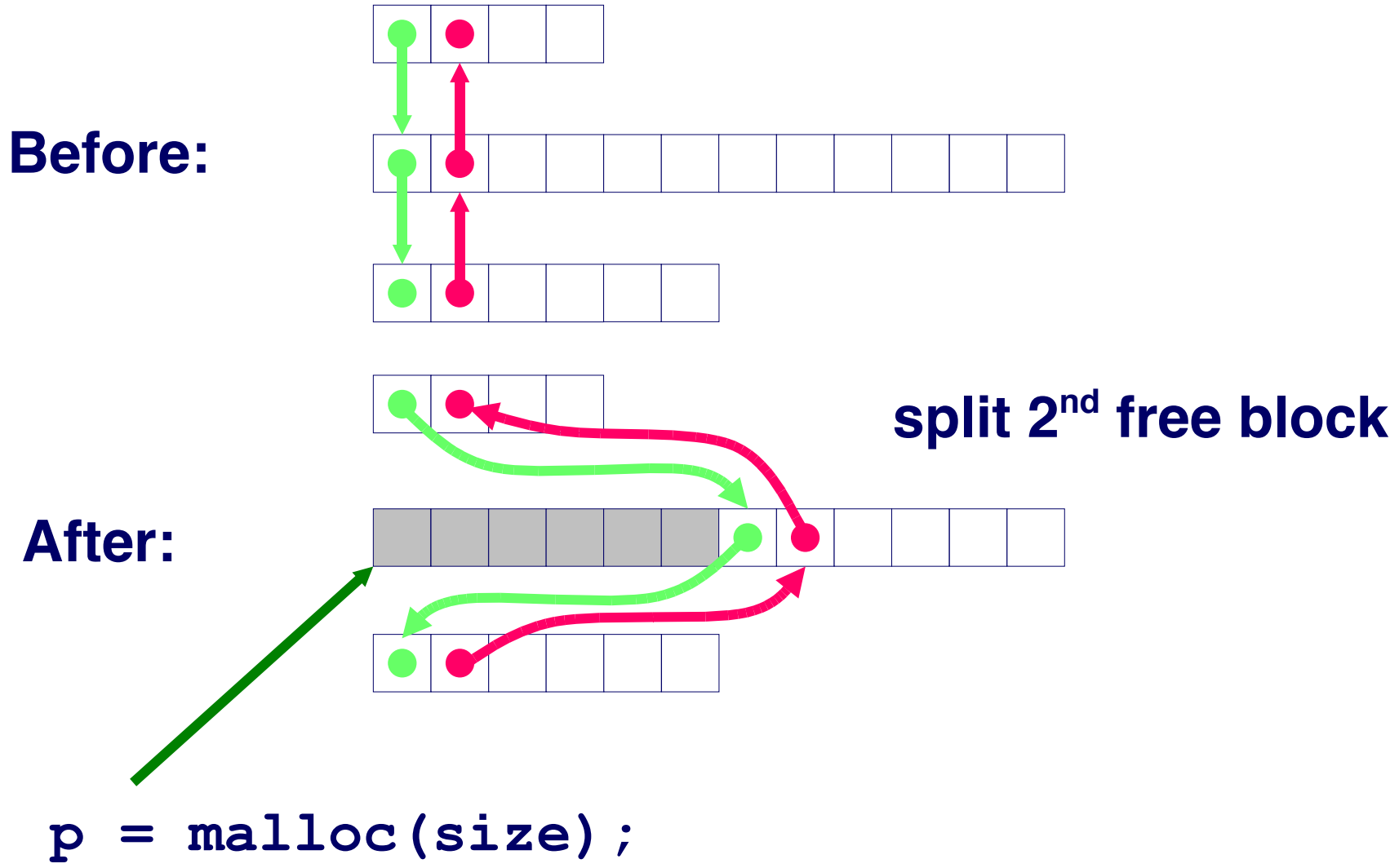
Explicit Free Lists

Note that free blocks need not be linked in the same order they appear in memory!

- Free blocks can be chained together in any order.



Allocation using an explicit free list



Deallocation with an explicit free list

Step 1: As before, mark block as free

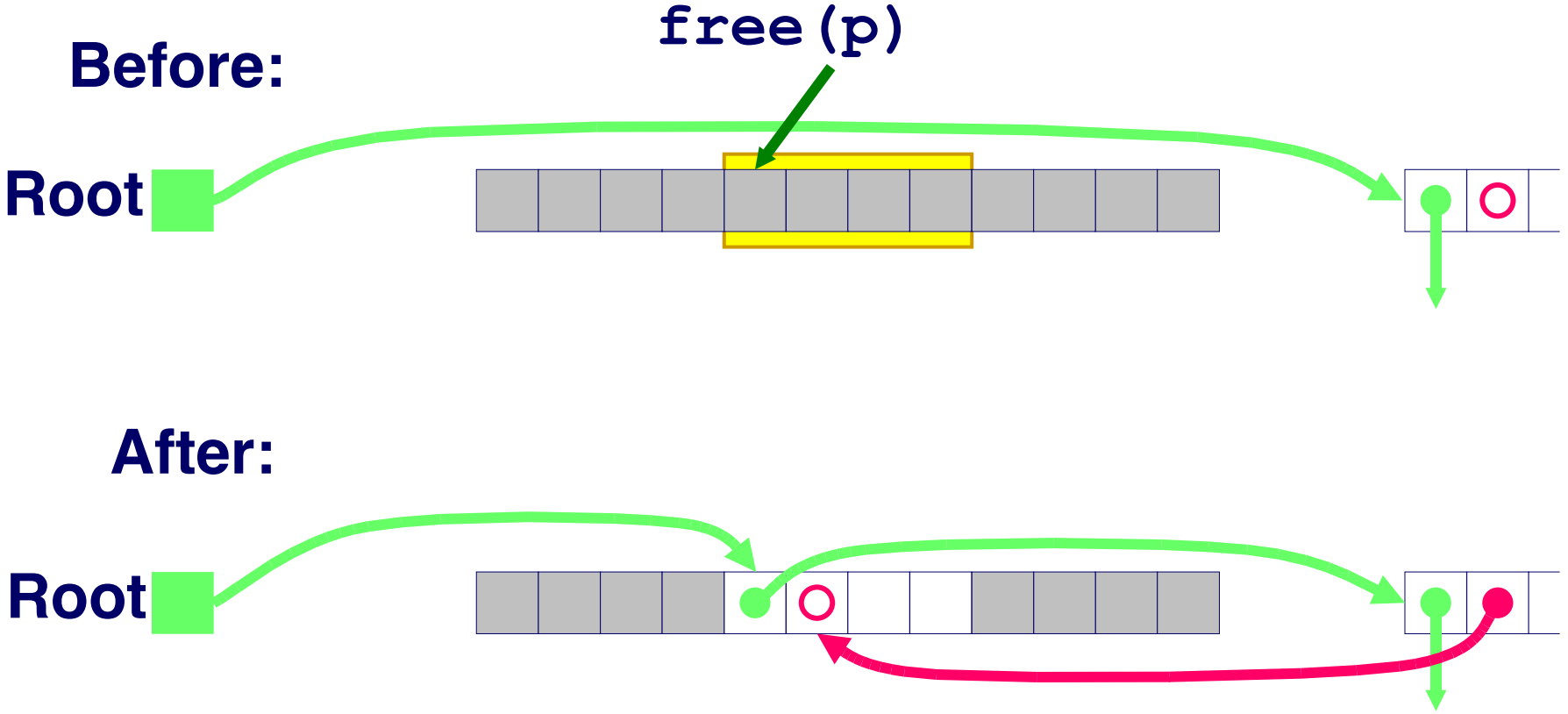
Step 2: Coalesce adjacent free blocks

Step 3: Insert free block into the free list

Where in the free list do we put a newly freed block?

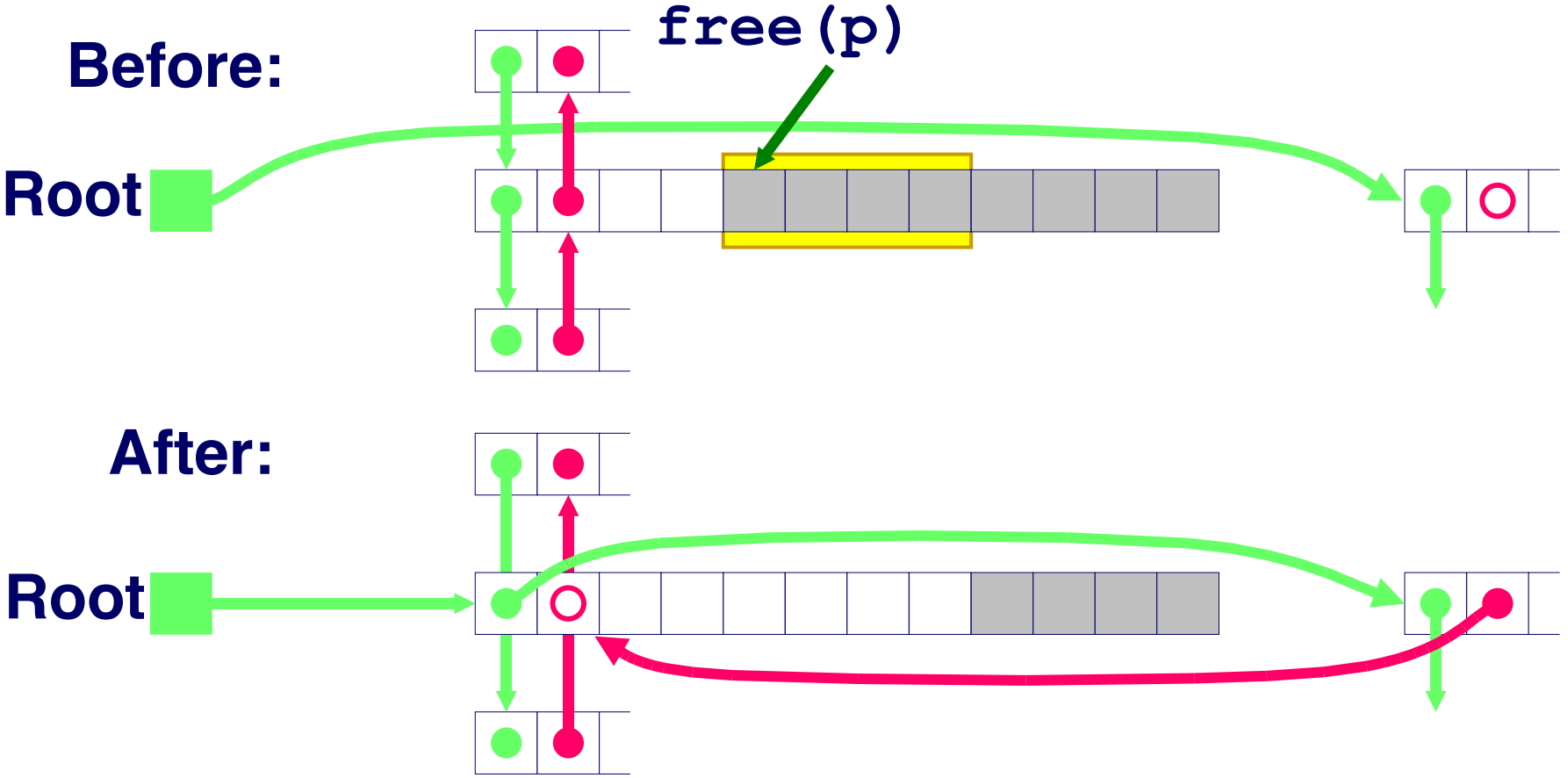
- LIFO (last-in-first-out) policy
  - *Insert freed block at the beginning of the free list*
  - *Simple and constant time*
- Address-ordered policy
  - *Insert freed blocks so that free list blocks are always in address order*
  - *Con: requires search*
  - *Pro: studies suggest fragmentation is lower than LIFO*

Freeing With a LIFO Policy (Case 1)



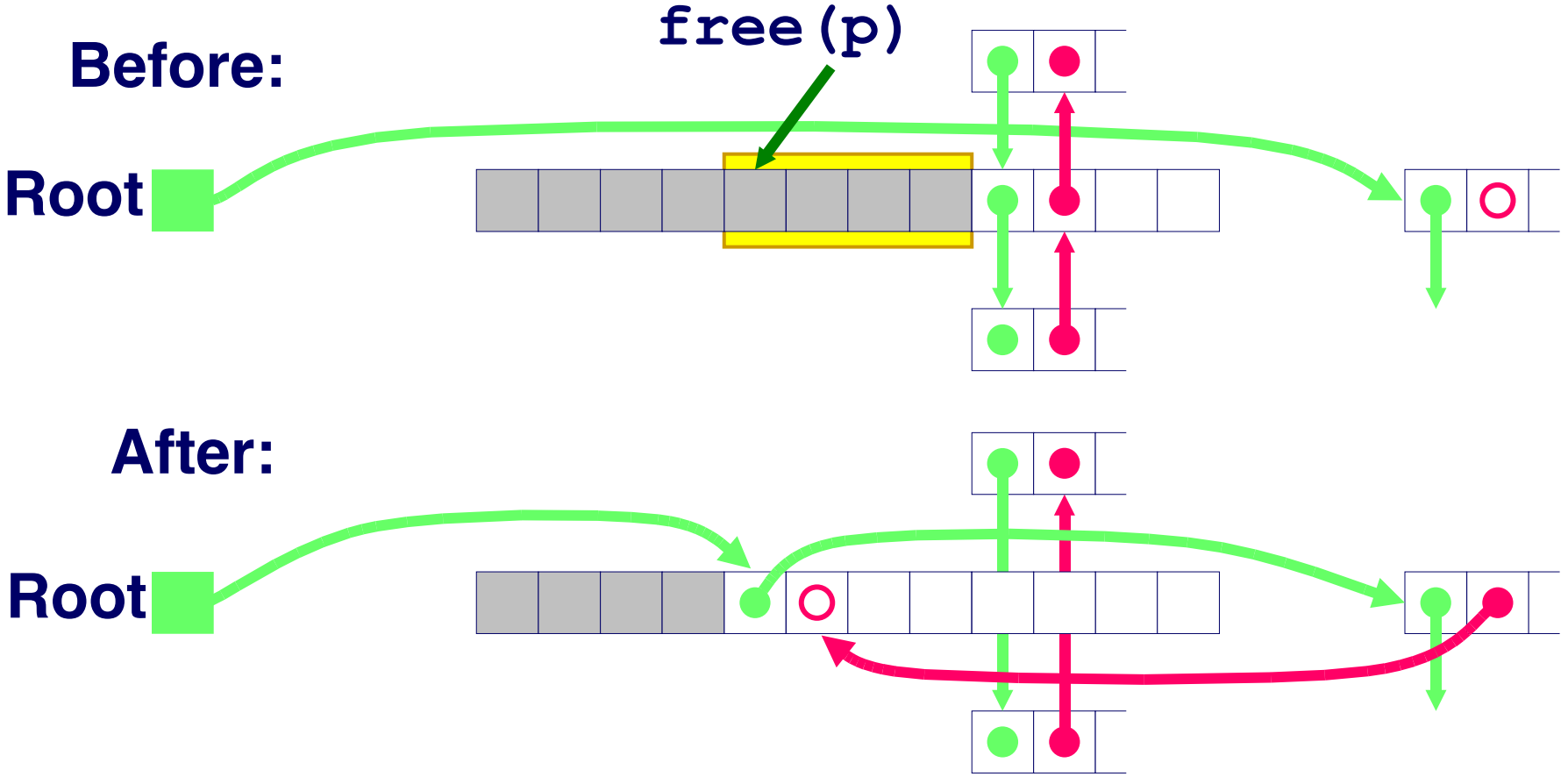
Insert the freed block at the root of the free block list

Freeing With a LIFO Policy (Case 2)



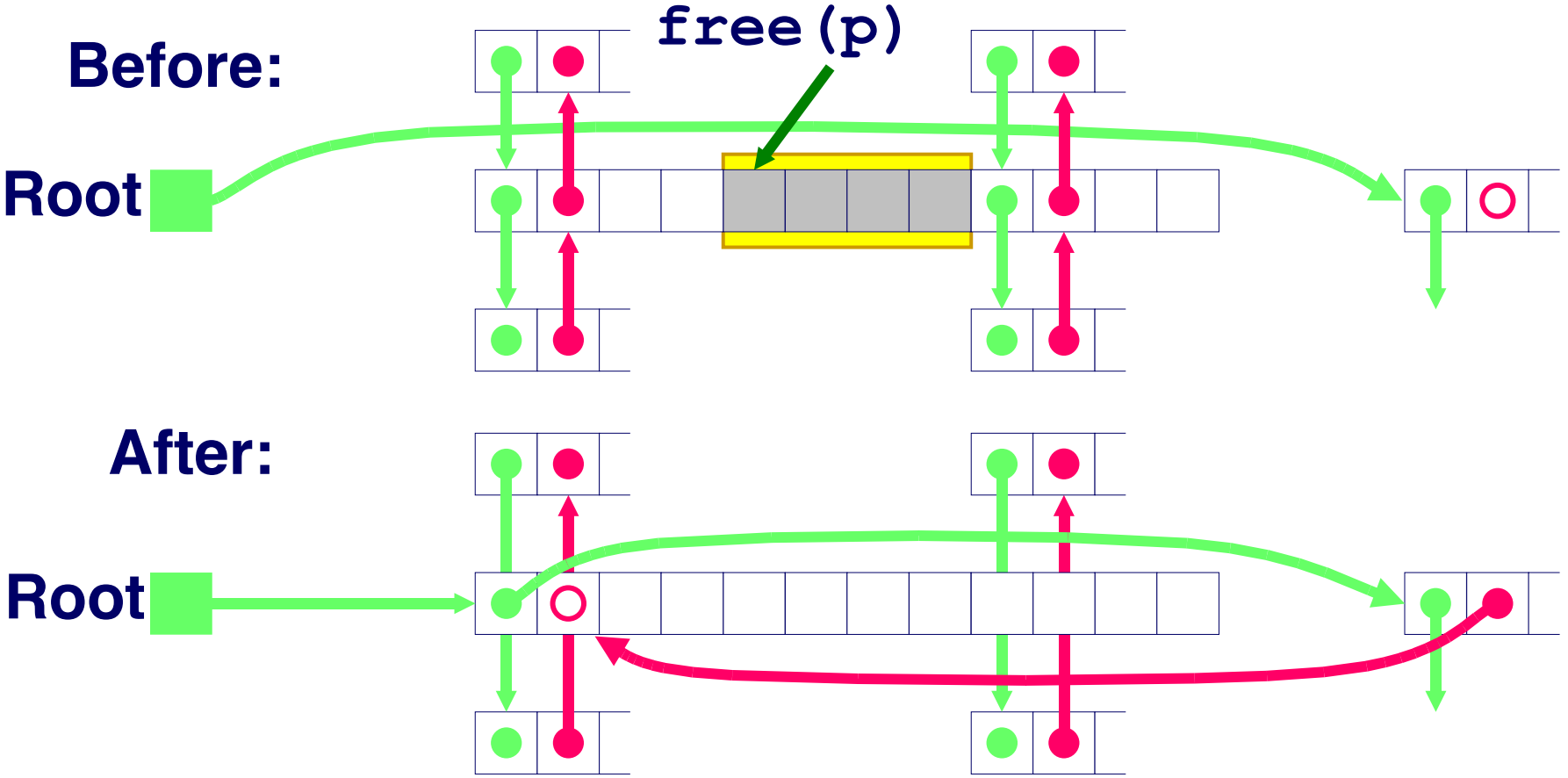
Splice out predecessor block, coalesce both memory blocks and insert the new block at the root of the list

Freeing With a LIFO Policy (Case 3)



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 4)



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:

- Allocation is linear time in number of **free** blocks
- Implicit list allocation is linear time in the number of **total** blocks
  - *Allocation for explicit lists is faster when memory is nearly full!*

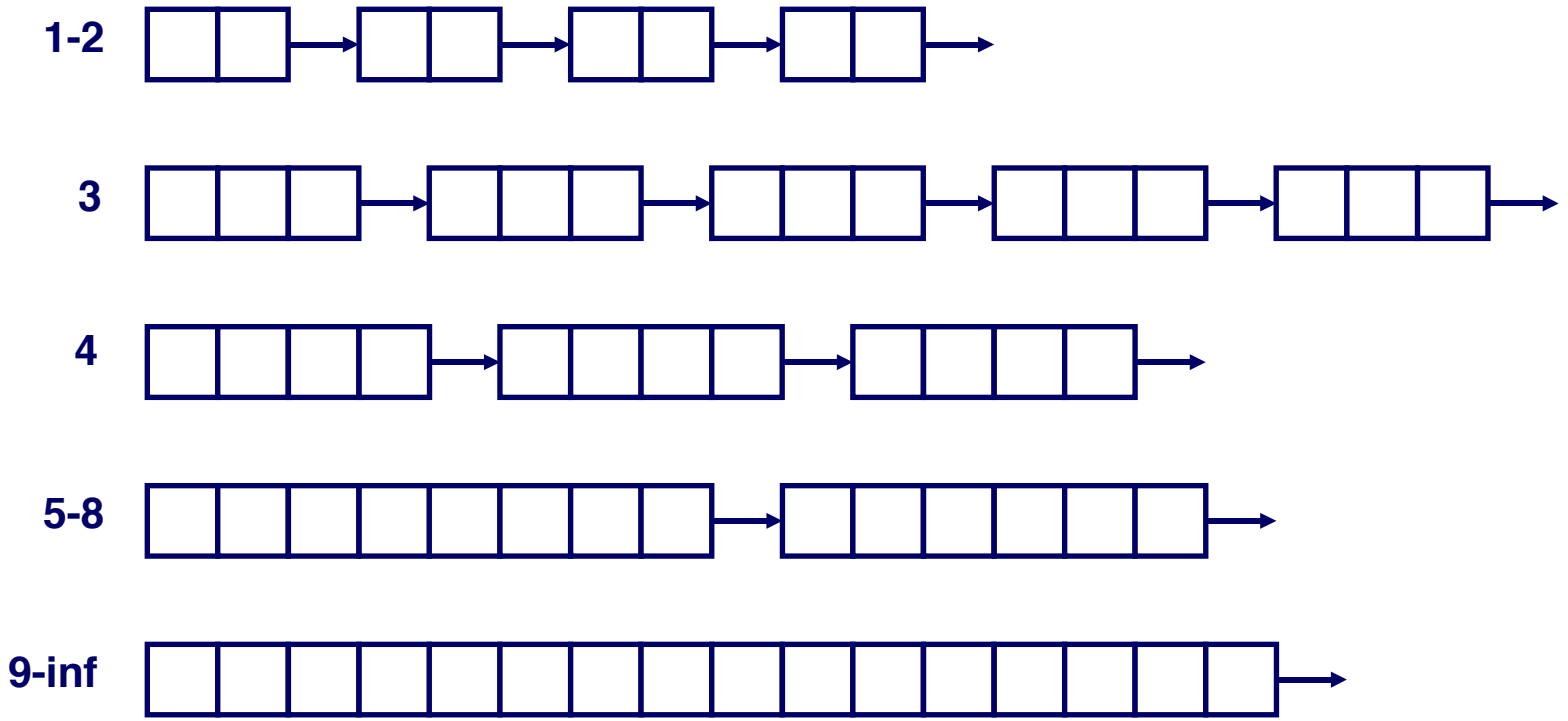
Slightly more complicated allocate and free since needs to splice blocks in and out of the list

Some extra space for the links (2 extra words needed for each free block) ...

- But these can be stored **in the payload**, since they are only needed for free blocks.

Segregated List (seglist) Allocators

Use a different free list for blocks of different sizes!



- Often have separate size class for every small size (4,5,6,…)
- For larger sizes typically have a size class for each power of 2

Seglist Allocator

To allocate a block of size *n*:

- Determine correct free list to use
- Search that free list for block of size *m* >= *n*
- If an appropriate block is found:
  - *Split block and place fragment on appropriate list*
- If no block is found, try next larger class
- Repeat until block is found

If no free block is found:

- Request additional heap memory from OS (using `sbrk()` system call)
- 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:

- Mark block as free
- Coalesce (if needed)
- Place free block on appropriate sized list

Advantages of seglist allocators

- Higher throughput
  - *Faster to find appropriate sized block: Look in the right list.*
- 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.*

Implicit Memory Management: Garbage Collection

Garbage collection: automatic reclamation of heap-allocated storage -- application never has to explicitly free memory!

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

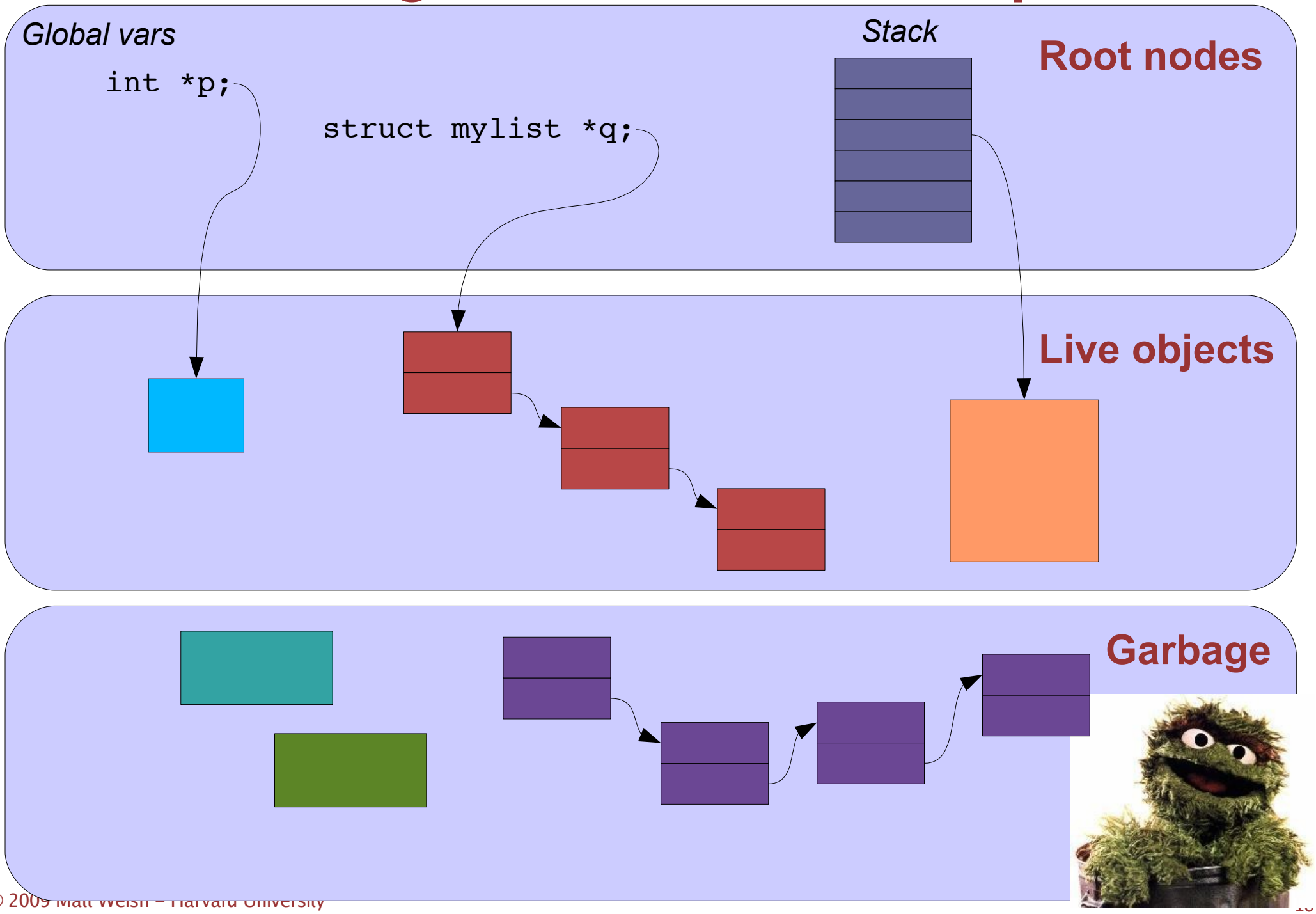
Common in functional languages, scripting languages, and modern object oriented languages:

- Lisp, ML, Java, Perl, Python, etc.

Variants (conservative garbage collectors) also exist for C and C++

- These do not generally manage to collect all garbage though.

Garbage collection concepts

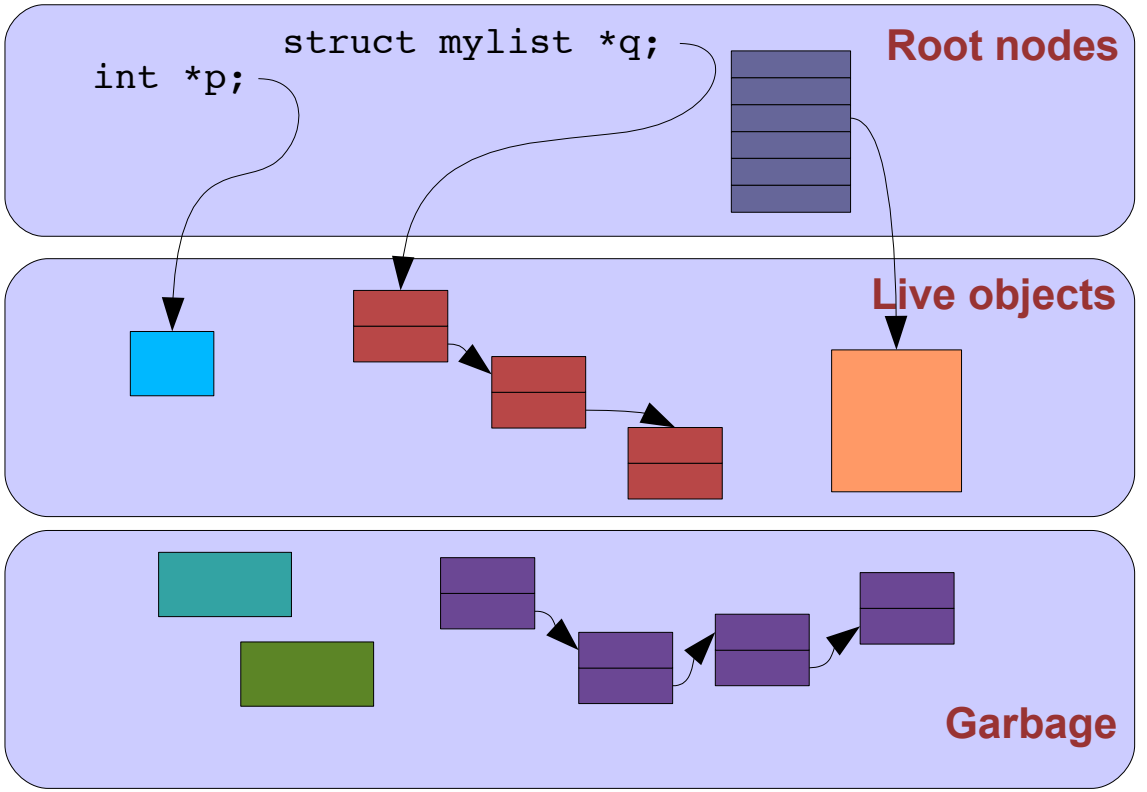


Garbage collection concepts

A block of memory is **reachable** if there is some path from a root node to the block.

- If a block is reachable, it's “live” and should not be reclaimed.
- If a block is not reachable, it's **garbage**.

Say we have a block that is reachable, but never used by the program. What happens?



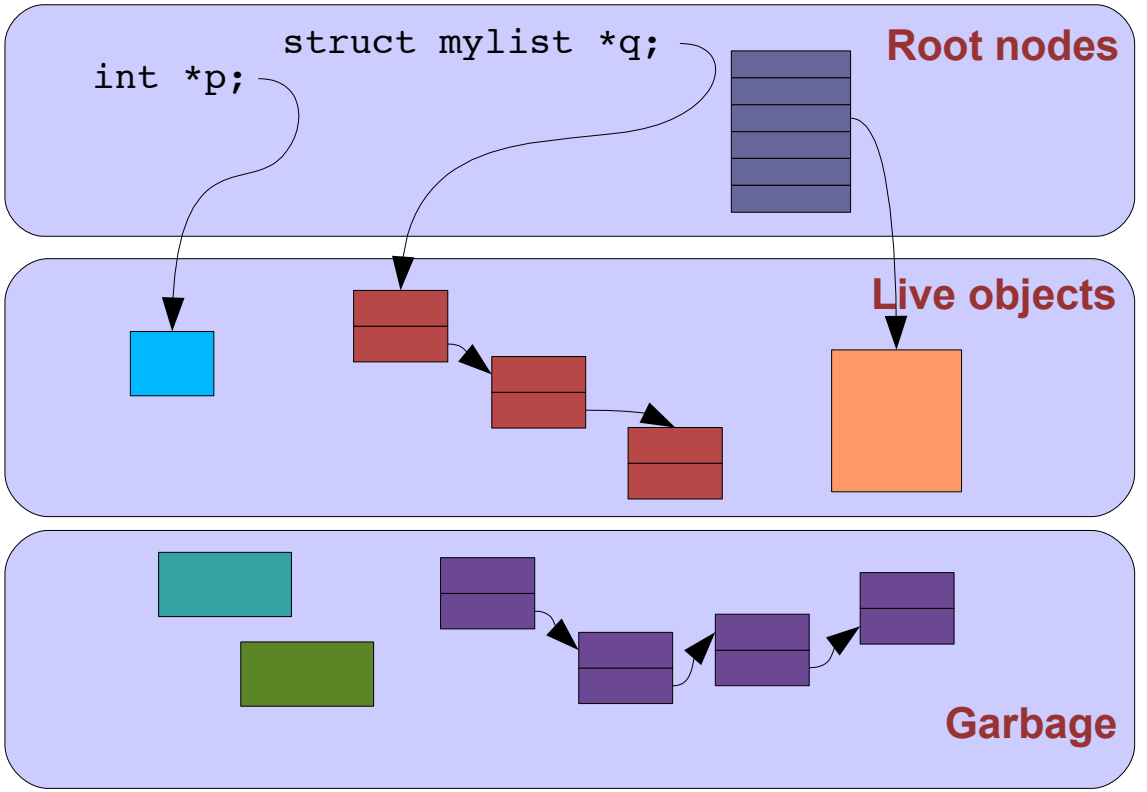
Garbage collection concepts

A block of memory is **reachable** if there is some path from a root node to the block.

- If a block is reachable, it's **live** and should not be reclaimed.
- If a block is not reachable, it's **garbage**.

Say we have a block that is reachable, but never used by the program. What happens?

- **Memory leak!**
- Garbage collectors assume all reachable objects can be used in the future.



Challenges

Challenge: How do we know where the pointers are?

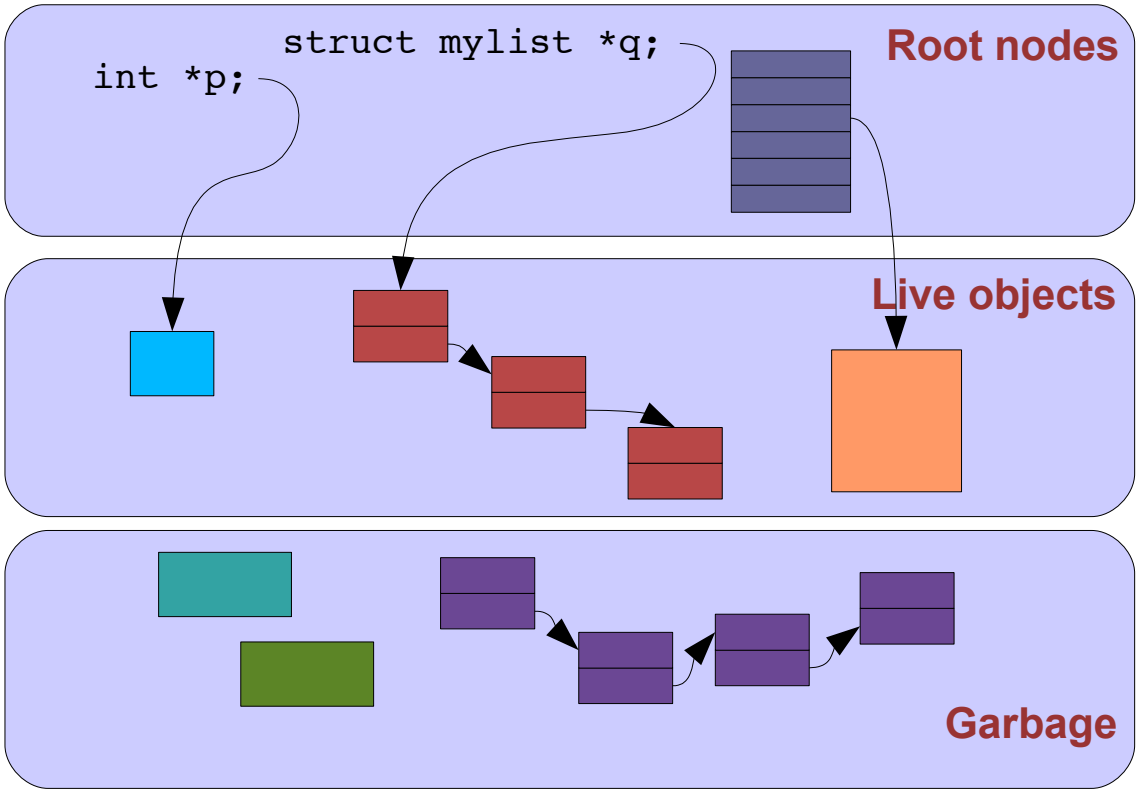
Global vars are easy: We know the type at compile time.

What about the stack?

- GC needs to know which slots on the stack are pointers, and which are not.
- Depends on the **call chain** of functions at runtime, and how each function stores local vars on the stack.

What about pointers *between* blocks of memory?

- Linked lists, trees, etc.
- GC needs to know where the pointers are internal to each block!



Fake pointers

- In C, you can create a pointer directly from an integer...

```
int i = 0x40b0cdf;
*(int *)i = 42;
```

- Problem: GC doesn't know that “i” is actually being used as a pointer.
  - Its type is simply “int”
- **Conservative garbage collectors** treat all bit patterns in memory that *could* be pointers as pointers.
  - If the contents of a word happen to correspond to a memory address within the heap, assume it is a pointer.
- Problem: Leads to “false” memory leaks
  - If I have “int i = 0x40b0cdf” but this is really just an integer value, not a pointer, whatever happens to be at memory location 0x40b0cdf will not be reclaimed!
  - Even worse: If the contents of memory at that location also appear to contain pointers, whatever they point to will not be reclaimed...
  - And so forth.

Fake pointers

- In C, you can create a pointer directly from an integer...

```
int i = 0x40b0cdf;
*(int *)i = 42;
```

- Problem: GC doesn't know that “i” is actually being used as a pointer.
  - Its type is simply “int”
- **Conservative garbage collectors** treat all bit patterns in memory that *could* be pointers as pointers.
  - If the contents of a word happen to correspond to a memory address within the heap, assume it is a pointer.
- Solution: Don't let program create pointers from ints.
  - Java (and other languages) never let you “forge” pointers like this.
  - Allows for **precise** garbage collection: GC always knows where pointers are, since pointers are “special”.

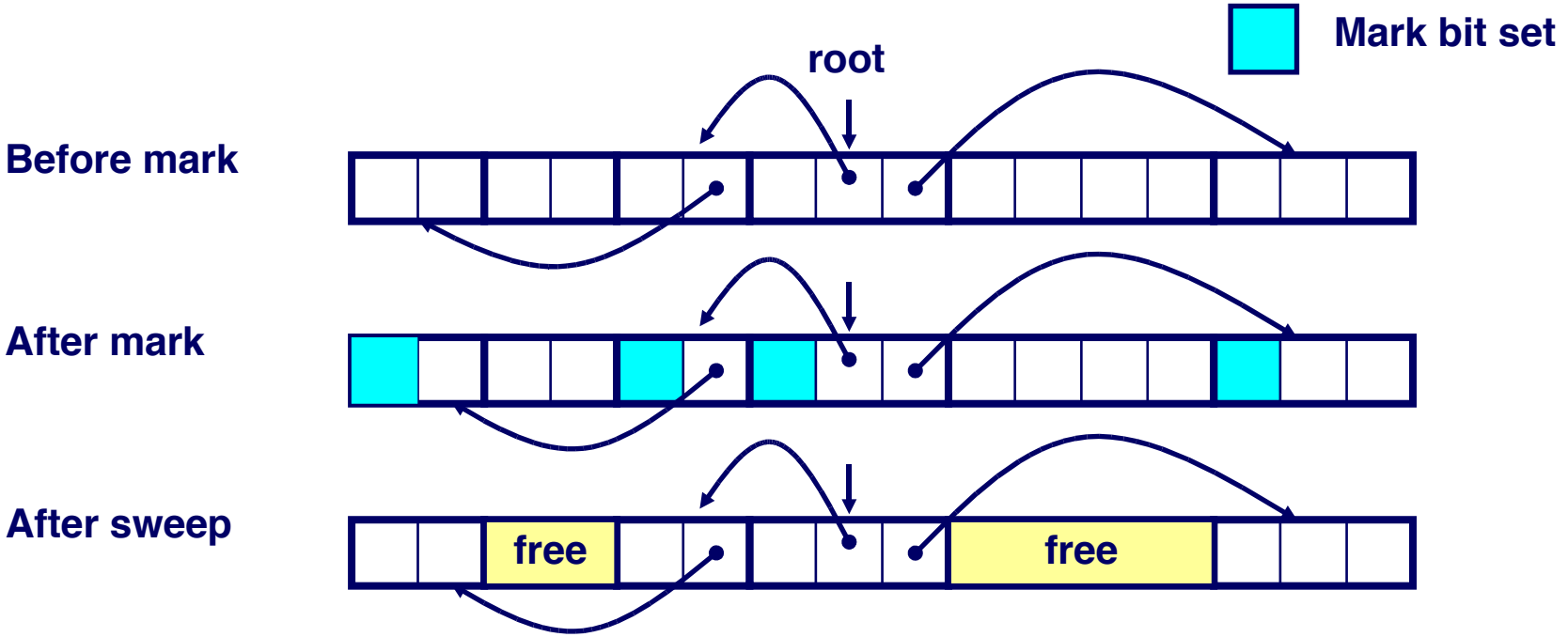
Mark and Sweep Collecting

Idea: Use a **mark bit** in the header of each block

- GC scans all memory objects (starting at roots), and sets mark bit on each reachable memory block.

Sweeping:

- Scan over all memory blocks.
- Any block without the mark bit set is freed



Other "classic" GC algorithms

Reference counting (Collins, 1960)

- Maintain a reference count for each memory block
- Adding a pointer to an object increases its refcount
- Removing a pointer decreases its refcount
- When refcount == 0, can collect.

Copying collection (Minsky, 1963)

- Find all of the live objects and copy them to a new location in memory.
- Everything else must be garbage, so free it.

Generational Collectors (Lieberman and Hewitt, 1983)

- Observation: Many objects live for short periods of time (i.e., local vars in a function).
- Scanning everything in memory is slow.
- Idea: If an object is live for a certain length of time, move it to a different part of the heap reserved for long-lived objects.
- Scan the "new" objects more frequently than the "old" objects.

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

Dereferencing Bad Pointers

The classic `scanf` bug

```
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

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 */
```

Overwriting Memory

Misunderstanding pointer arithmetic

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

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

   return p;
}
```

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);
```

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

Conventional debugger (`gdb`)

- Good for finding bad pointer dereferences
- Hard to detect the other memory bugs

Debugging `malloc` (CSRI UToronto `malloc`)

- Wrapper around conventional `malloc`
- Detects memory bugs at `malloc` and `free` boundaries
  - *Memory overwrites that corrupt heap structures*
  - *Some instances of freeing blocks multiple times*
  - *Memory leaks*
- Cannot detect all memory bugs
  - *Overwrites into the middle of allocated blocks*
  - *Freeing block twice that has been reallocated in the interim*
  - *Referencing freed blocks*

Dealing With Memory Bugs (cont.)

Binary translator: valgrind (Linux), Purify

- Powerful debugging and analysis technique
- Rewrites text section of executable object file
- Can detect all errors as debugging `malloc`
- Can also check each individual reference at runtime
  - *Bad pointers*
  - *Overwriting*
  - *Referencing outside of allocated block*

Garbage collection (Boehm-Weiser Conservative GC)

- Let the system free blocks instead of the programmer.

Review of C Pointer Declarations

`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