

# 15-213

## Dynamic Memory Allocation II

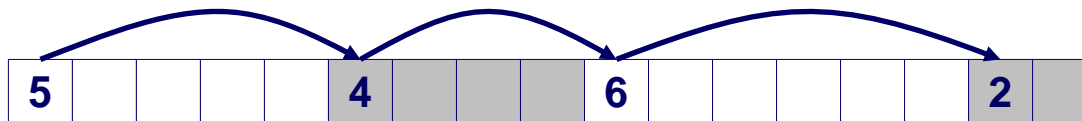
### October 22, 2008

#### Topics

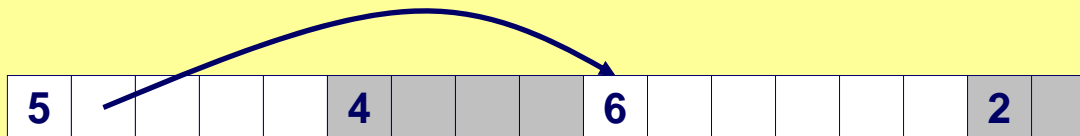
- Explicit doubly-linked free lists
- Segregated free lists
- Garbage collection
- Review of pointers
- Memory-related perils and pitfalls

# Keeping Track of Free Blocks

- **Method 1**: Implicit list using lengths -- links all blocks



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



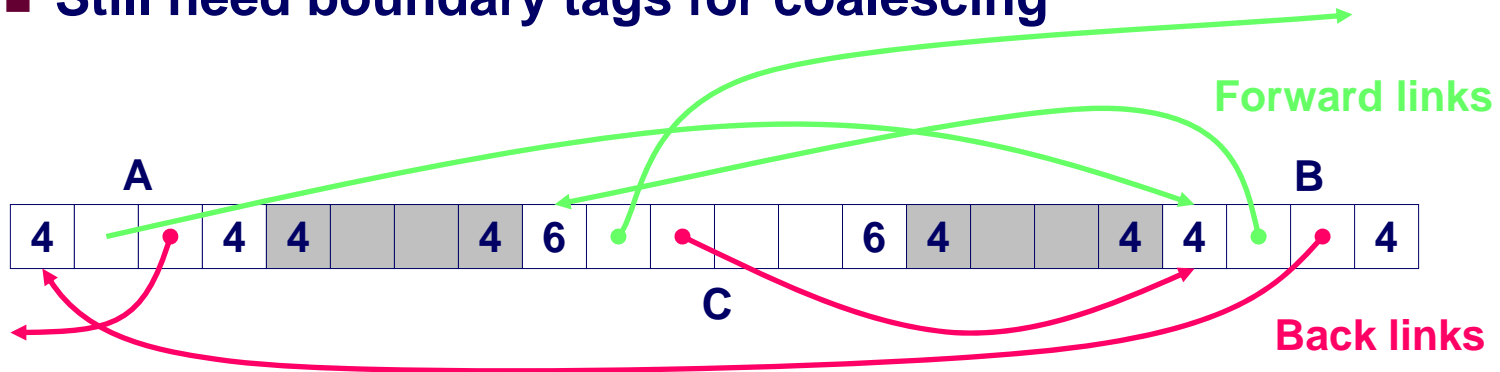
- **Method 3**: Segregated free lists
  - Different free lists for different size classes
- **Method 4**: Blocks sorted by size (not discussed)
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

# Explicit Free Lists



## Use data space for link pointers

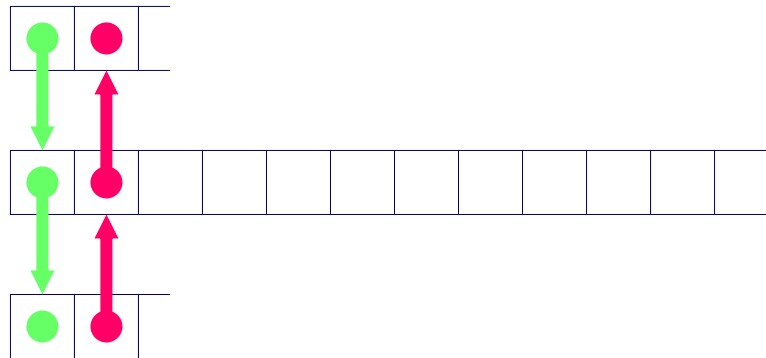
- Typically doubly linked
- Still need boundary tags for coalescing



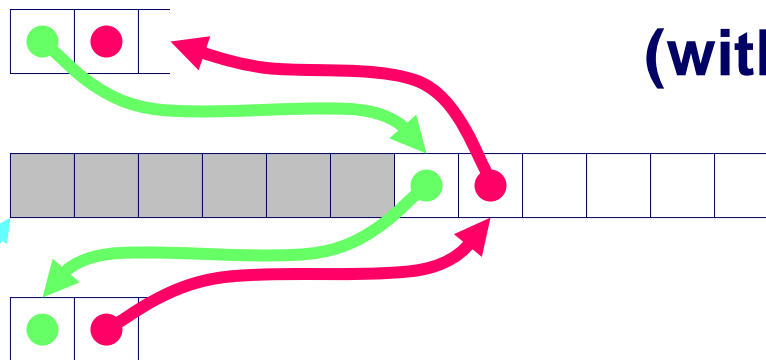
- It is important to realize that links are not necessarily in the same order as the blocks

# Allocating From Explicit Free Lists

Before:



After:



(with splitting)

● = `malloc(...)`

# Freeing With Explicit Free Lists

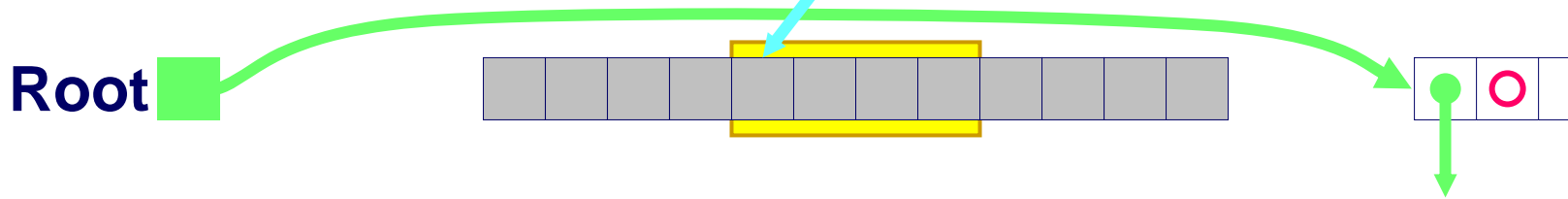
***Insertion policy:*** Where in the free list do you put a newly freed block?

- **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
    - » i.e.  $\text{addr}(\text{pred}) < \text{addr}(\text{curr}) < \text{addr}(\text{succ})$
  - Con: requires search
  - Pro: studies suggest fragmentation is lower than LIFO

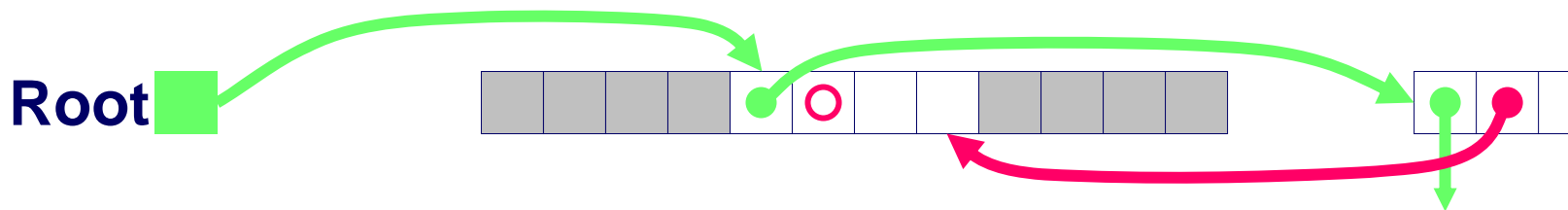
# Freeing With a LIFO Policy (Case 1)

free ( ● )

Before:

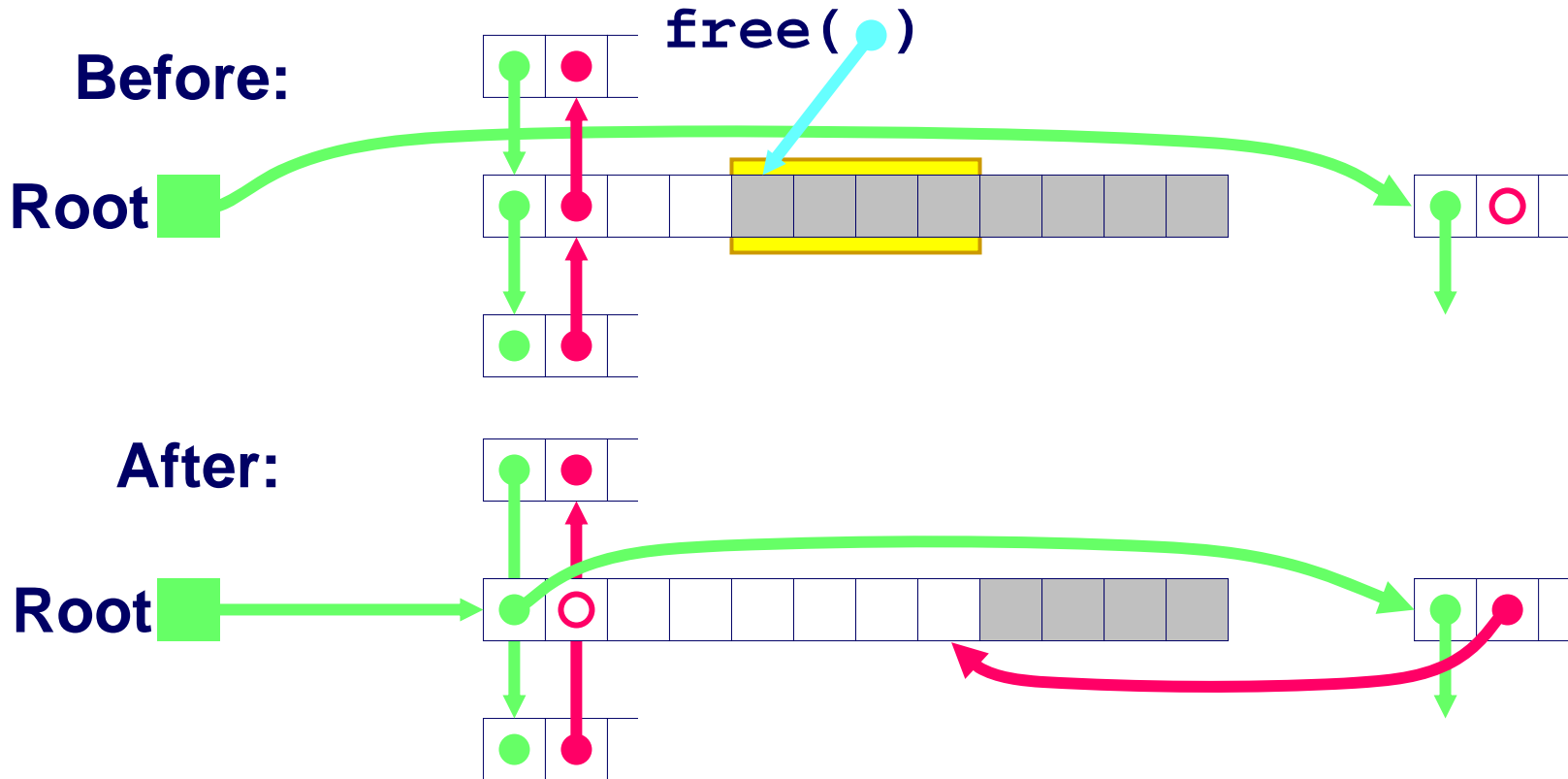


After:



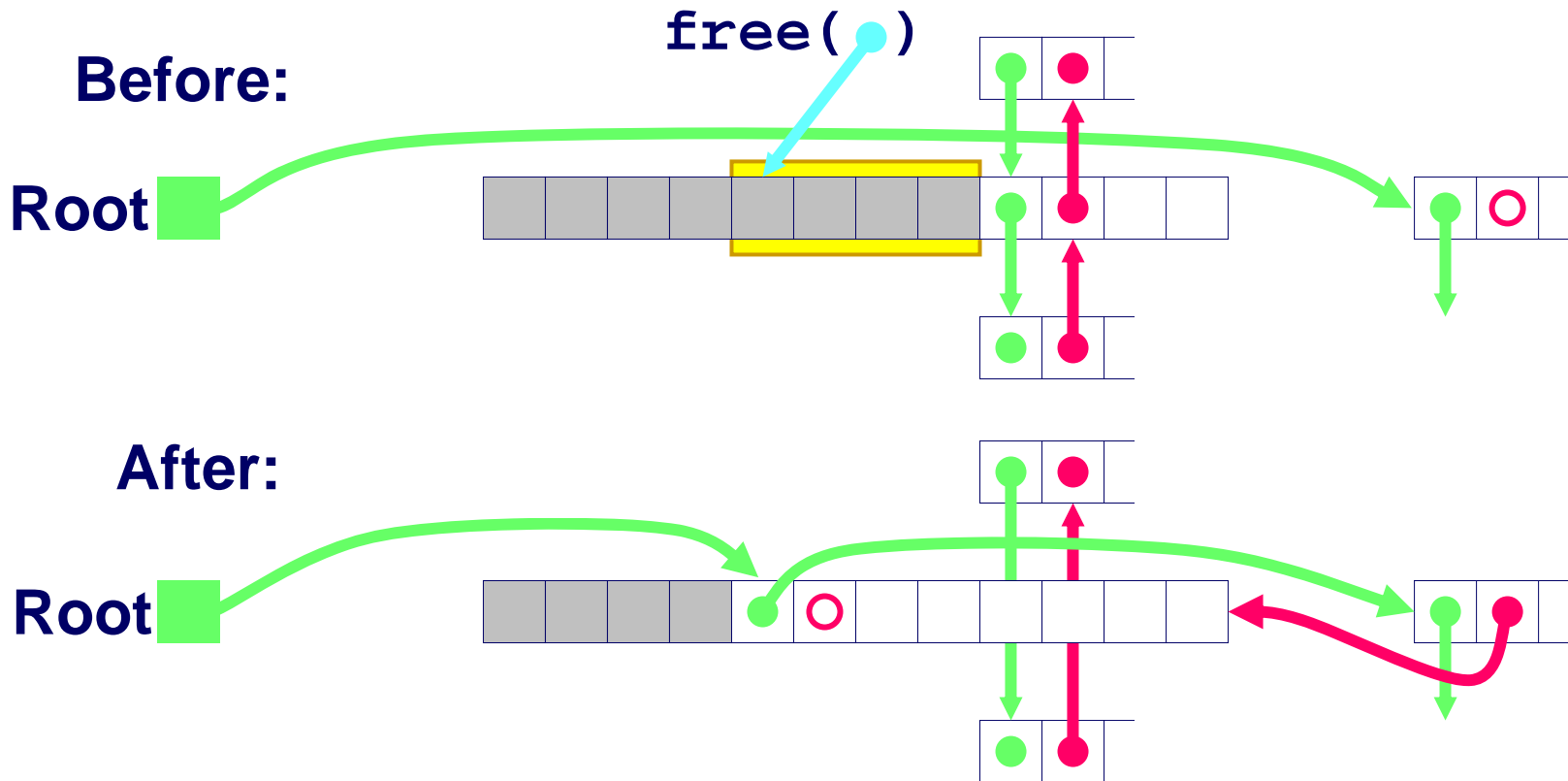
Insert the freed block at the root of the list

# 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

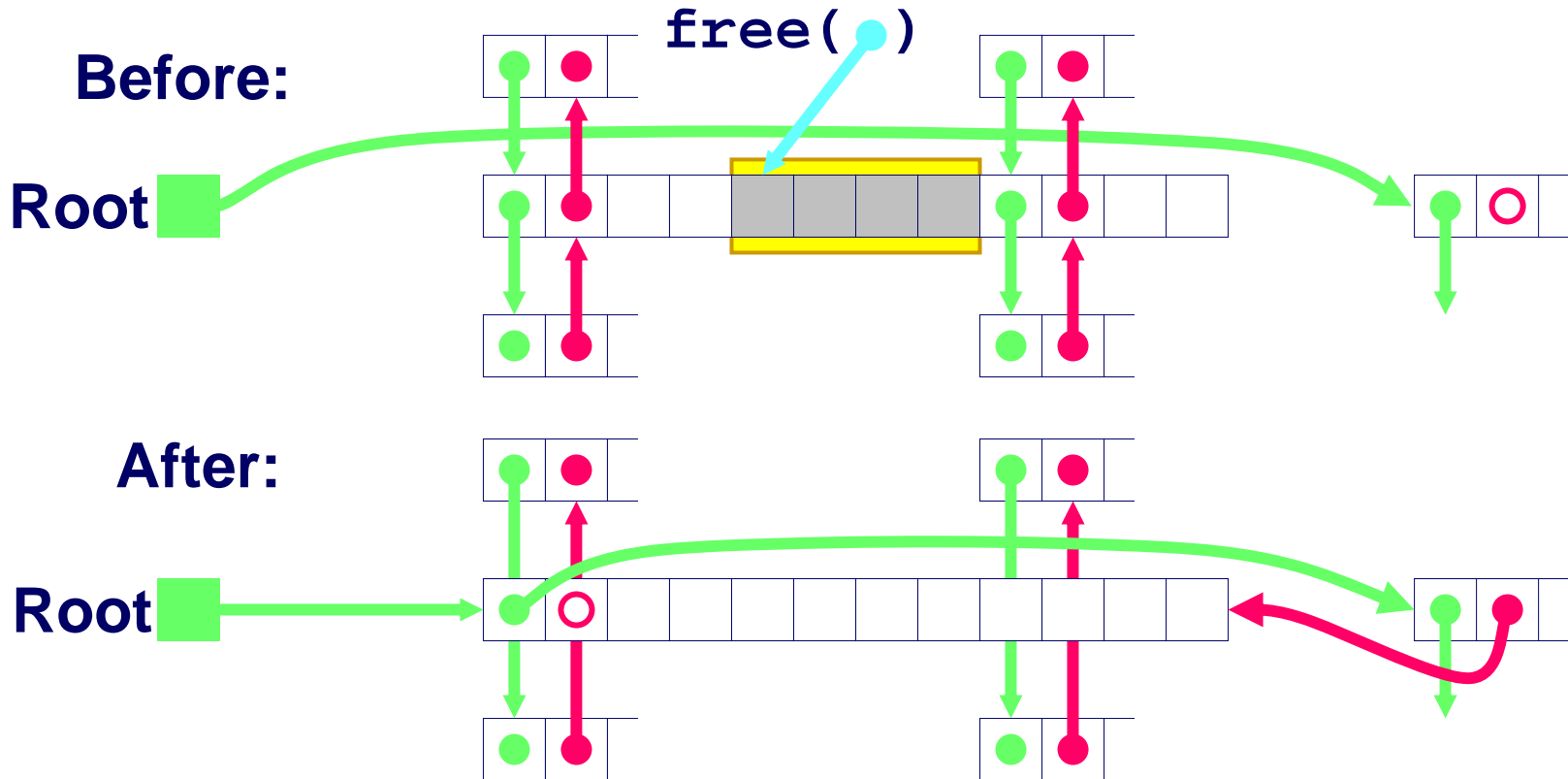
# Freeing With a LIFO Policy (Case 3)



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:

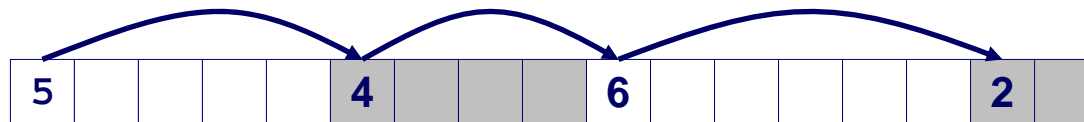
- Allocate is linear time in number of free blocks instead of total blocks --
  - much faster allocates when most of the memory is full
- Slightly more complicated allocate and free since needs to splice blocks in and out of the list
- Some extra space for the links (2 extra words needed for each block)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

## Main use of linked lists is in conjunction with segregated free lists

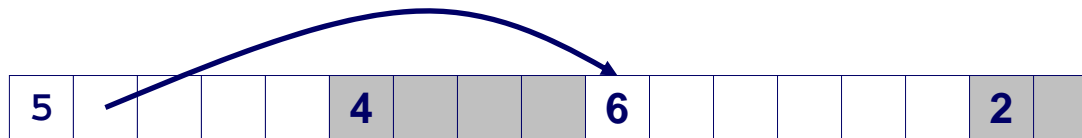
- Keep multiple linked lists of different size classes, or possibly for different types of objects

# Keeping Track of Free Blocks

**Method 1: Implicit list** using lengths -- links all blocks



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



**Method 3: Segregated free list**

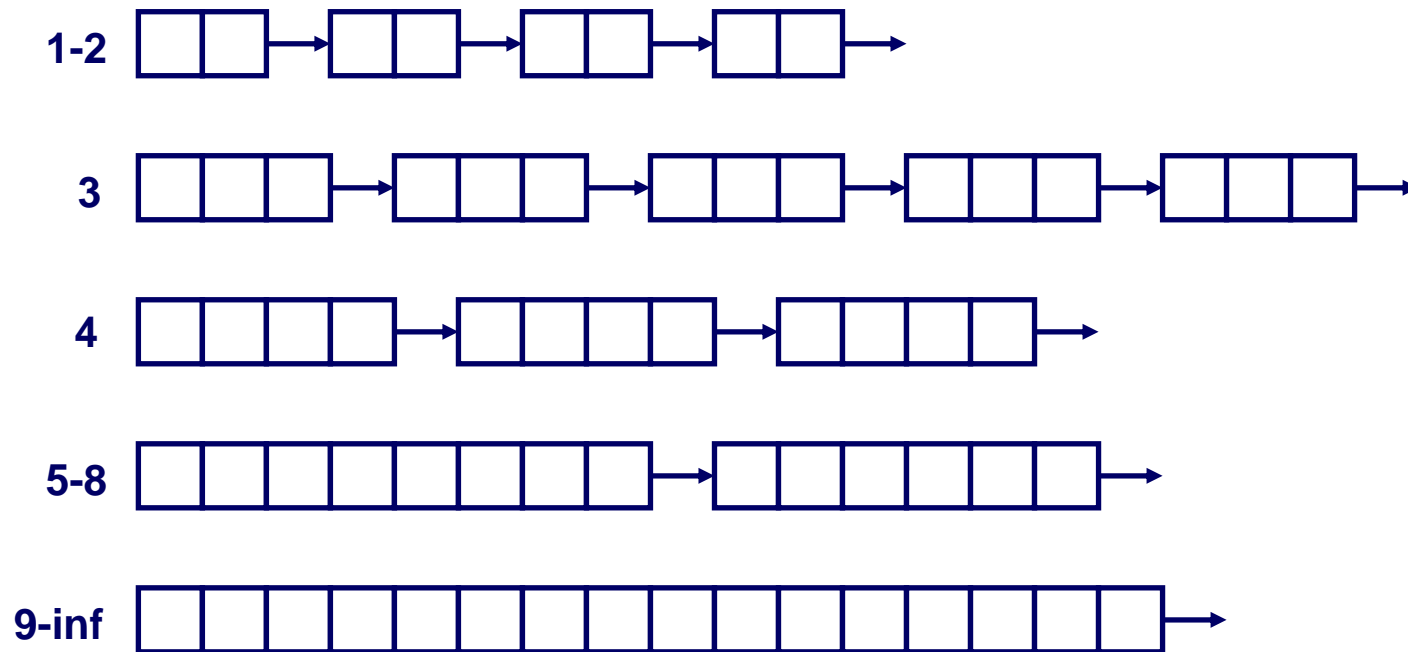
- Different free lists for different size classes

**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

# Segregated List (seglis) Allocators

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



- Often have separate size class for every small size (2,3,4,...)
- For larger sizes typically have a size class for each power of 2

# 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` function)
- 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 (optional)

## Advantages of seglist allocators

- Higher throughput
  - i.e., 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.

# For More Info on Allocators

D. Knuth, “*The Art of Computer Programming, Second 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](http://csapp.cs.cmu.edu))

# 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 functional languages, scripting languages, and modern object oriented languages:

- Lisp, ML, Java, Perl, 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

## Need to 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)

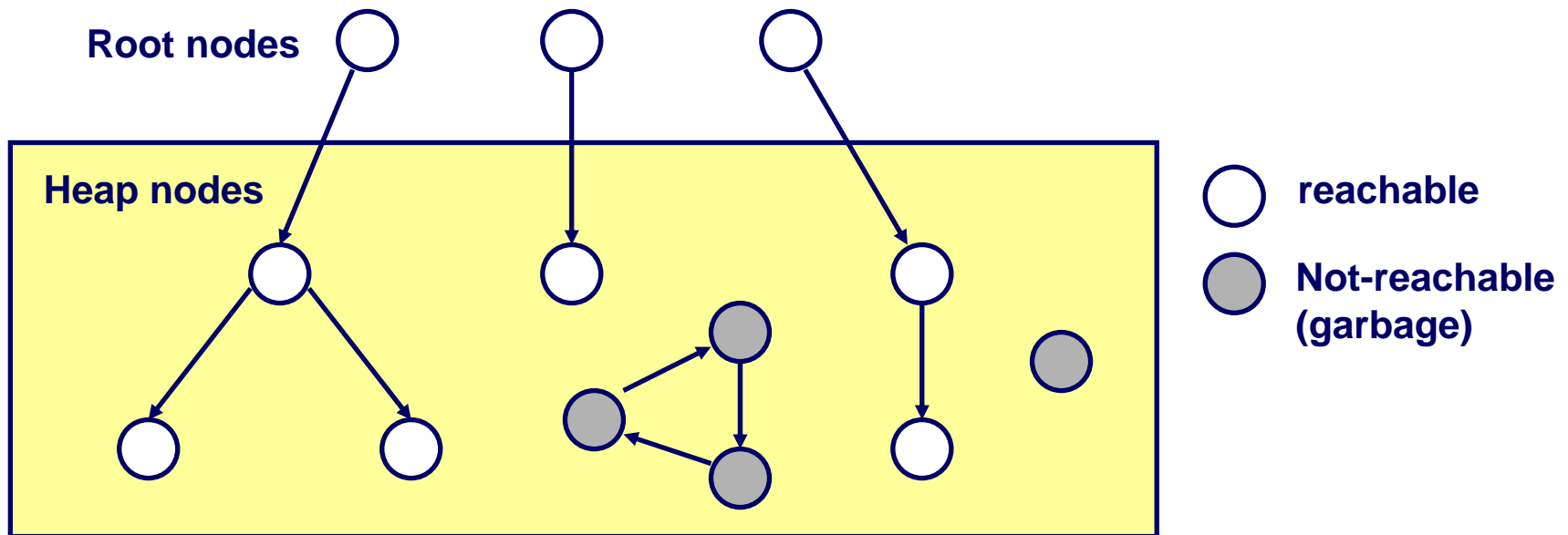
- Collects based on lifetimes

For more information, see 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** (never needed by the application)

# Assumptions For This Lecture

## 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

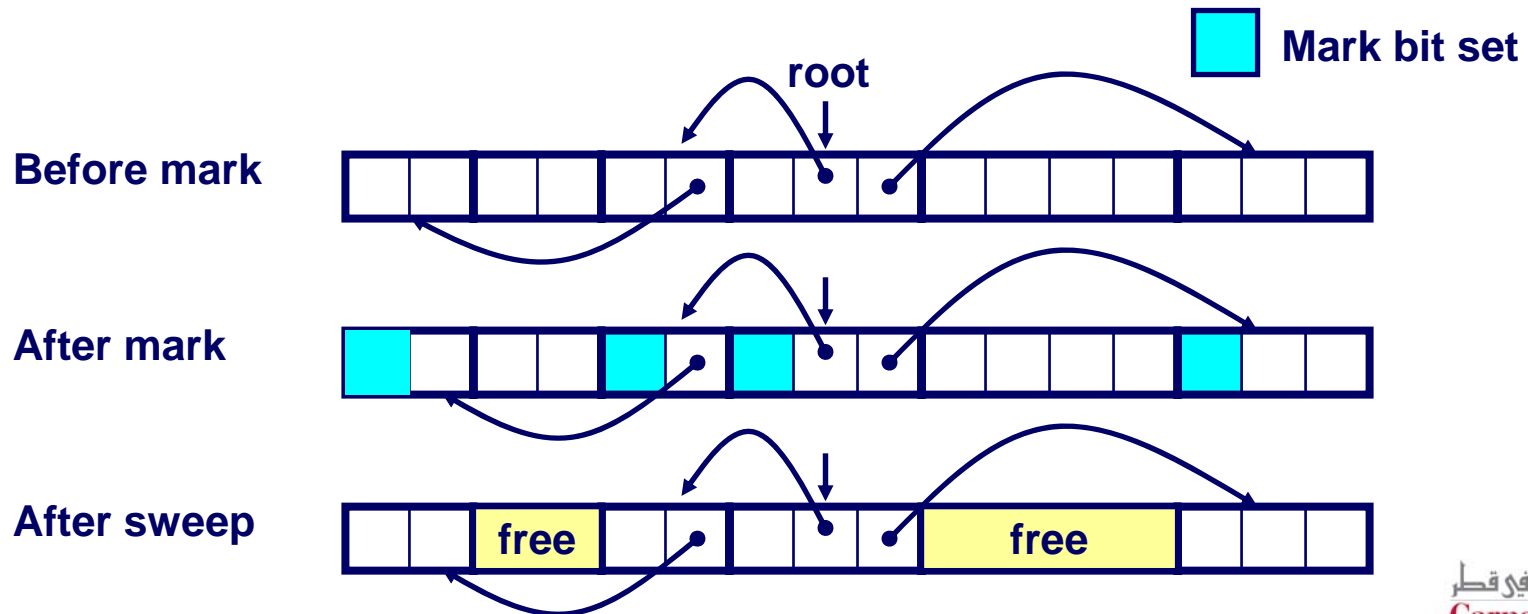
# Mark and Sweep Collecting

## 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 sets **mark bit** on all reachable memory
- **Sweep**: Scan all blocks and **free** blocks that are **not marked**



# 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++)   // mark all children  
        mark(p[i]);  
    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 and Sweep in C

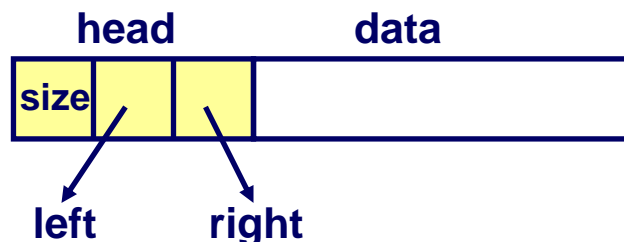
## A conservative collector for C programs

- `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 do we find the beginning of the block?

- Can use balanced tree to keep track of all allocated blocks where the key is the location
- Balanced tree pointers can be stored in header (use two additional words)



# 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 (K&R p. 53)

## Operators

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

## Associativity

left to right  
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  
right to left  
right to left  
left to right

**Note: Unary +, -, and \* have higher precedence than binary forms**

# Review of C Pointer Declarations

<code>int *p</code>	<code>p</code> is a pointer to <code>int</code>
<code>int *p[13]</code>	<code>p</code> is an array[13] of pointer to <code>int</code>
<code>int *(p[13])</code>	<code>p</code> is an array[13] of pointer to <code>int</code>
<code>int **p</code>	<code>p</code> is a pointer to a pointer to an <code>int</code>
<code>int (*p)[13]</code>	<code>p</code> is a pointer to an array[13] of <code>int</code>
<code>int *f()</code>	<code>f</code> is a function returning a pointer to <code>int</code>
<code>int (*f)()</code>	<code>f</code> is a pointer to a function returning <code>int</code>
<code>int ((*f())[13])()</code>	<code>f</code> is a function returning ptr to an array[13] of pointers to functions returning <code>int</code>
<code>int ((*x[3])())[5]</code>	<code>x</code> is an array[3] of pointers to functions returning pointers to array[5] of <code>ints</code>

# 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

## 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

- 1988 Internet worm
- Modern attacks on Web servers
- AOL/Microsoft IM war

# 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);  
}
```



# 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.