Chapter 9: Heap Management

Topics

- Implicit Free List
- Explicit Free List
- Segregated Free List

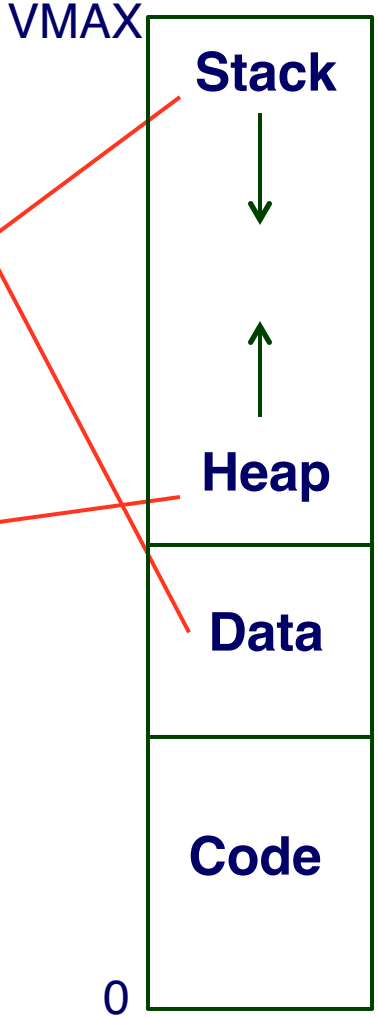
Announcements

- Shell lab grading this week
- Last Recitation Exercise #5 due Friday Dec 12 by 5 pm
  - Upload to moodle or hand into TA at TA office hours
- Final exam is Thursday Dec 18, 4:30-7 pm
  - Similar in format to midterms, closed book, no electronics, can bring 1 page front/back summary sheet and 5-page printouts
  - Comprehensive but mostly focused on Ch 6-9 plus a problem on C/assembly fill-in-blank plus a problem on Ch 5 optimization, so it will be longer than midterms
  - We'll release some practice problems later this week
- Reading:
  - Read Chapter 9, except 9.6 and 9.7
  - Read Chapter 7, except 7.12 (no position independent code)

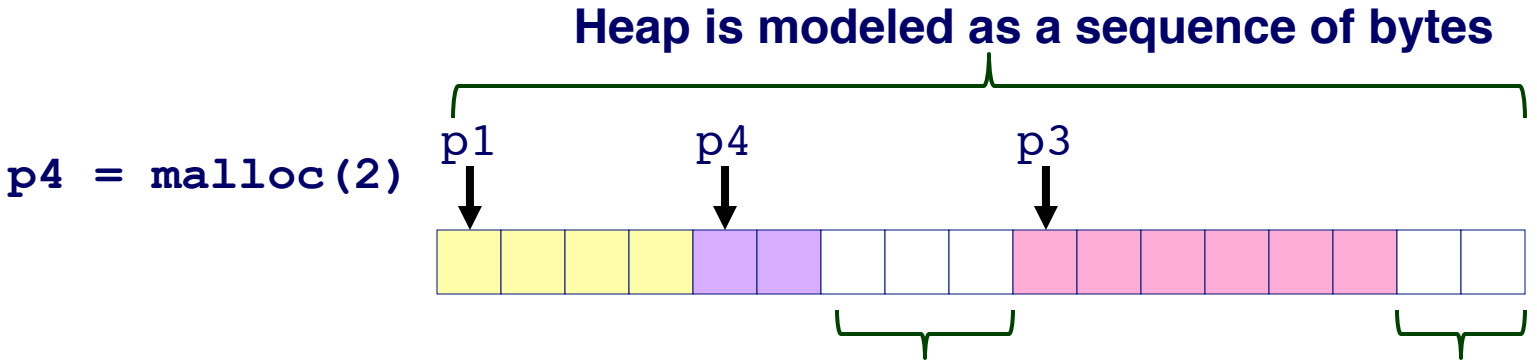
Recap: Heap Allocation Example

```
int x=0,y[1000];
char *p;

main (int argc, char *argv[]) {

     p = (char*) malloc(256);

     function1(p);

     free(p); /* return p to
available memory pool */

}

function1 (char *ptr) {

int i,j=50;

     for (i=0; i<100; i++) {
        *(ptr+i) = i*j;
     }

}
```

- Global variables allocated in data section of address space
- Local variables allocated on the stack
- Dynamic variables allocated on the heap
  - In C++, the "new" command is equivalent to malloc



Recap: Allocation Examples

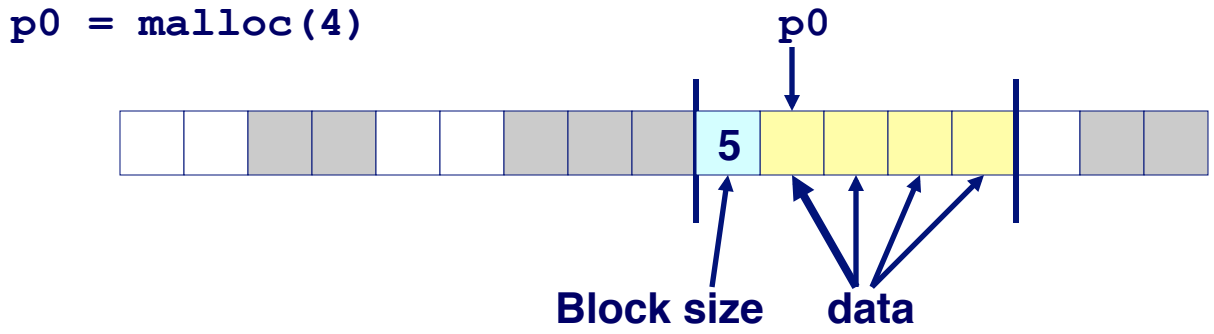


Fragmentation wastes space and may prevent allocation of big new blocks

Primary goals

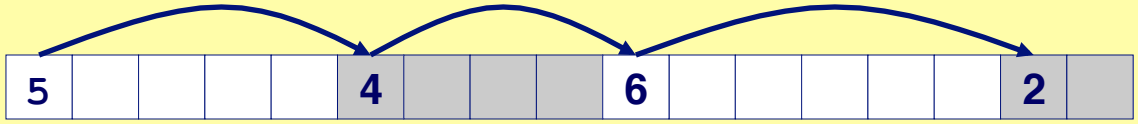
1. Efficient memory utilization
2. Low latency/fast throughput for `malloc` and `free`

- Goals 1 and 2 are often conflicting!

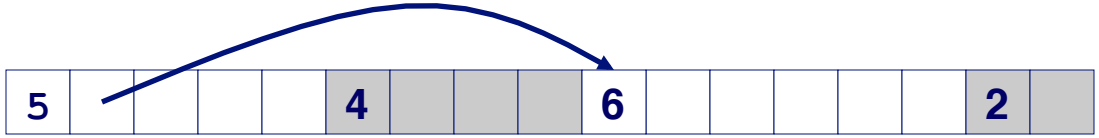


Keeping Track of Free Blocks

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



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



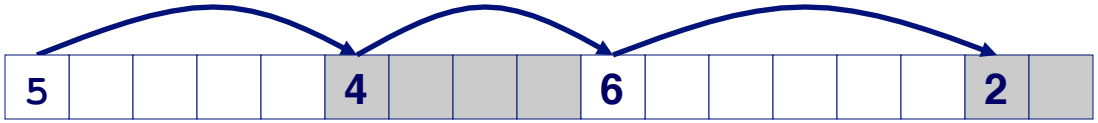
Method 3: ***Segregated free list***

- 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

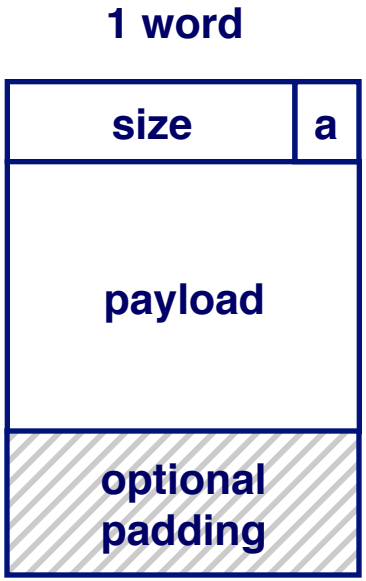
Method 1: Implicit List



Need to identify whether each block is free or allocated

- Can use extra bit
- Bit can be put in the same word as the size if block sizes are always multiples of two (mask out low order bit when reading size).

Format of allocated and free blocks

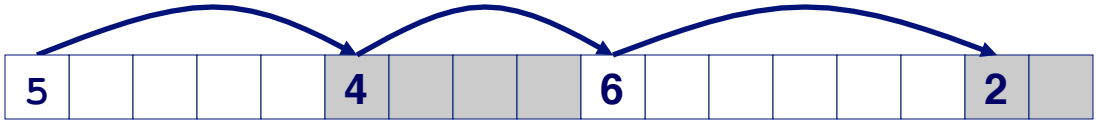


a = 1: allocated block
a = 0: free block

size: block size

payload: application data (allocated blocks only)

Implicit List: Finding a Free Block



First fit:

- Search list from beginning, choose first free block that fits

pseudo code

```
p = start;
while ((p < end) &&     \\ not passed end
       ((*p & 1) ||      \\ already allocated
       (!(*p & 1) && (*p <= len)));   \\ free but too small
       p += *p;          \\ not shown, mask alloc bit
```

- Can take linear time in total number of blocks (allocated and free)
- In practice it can cause “splinters” at beginning of list

Next fit:

- Like first-fit, but search list from location of end of previous search
- Research suggests that fragmentation is worse

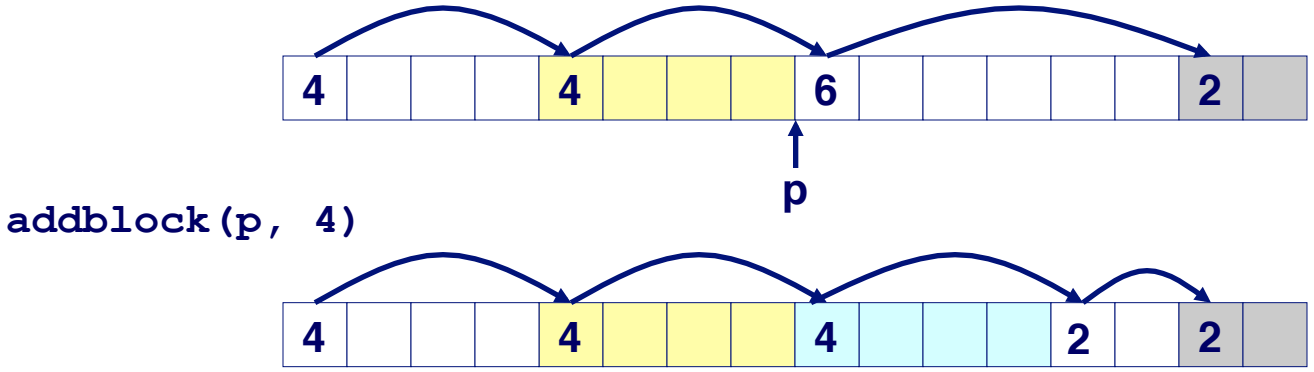
Best fit:

- Search the list, choose the free block with the closest size that fits
- Keeps fragments small --- usually helps fragmentation
- Will typically run slower than first-fit

Implicit List: Allocating in Free Block

Underallocating in a free block - *splitting*

- Since allocated space might be smaller than free space, we might want to split the block




```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // add 1 and round up
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining
}                                       //   part of block
```

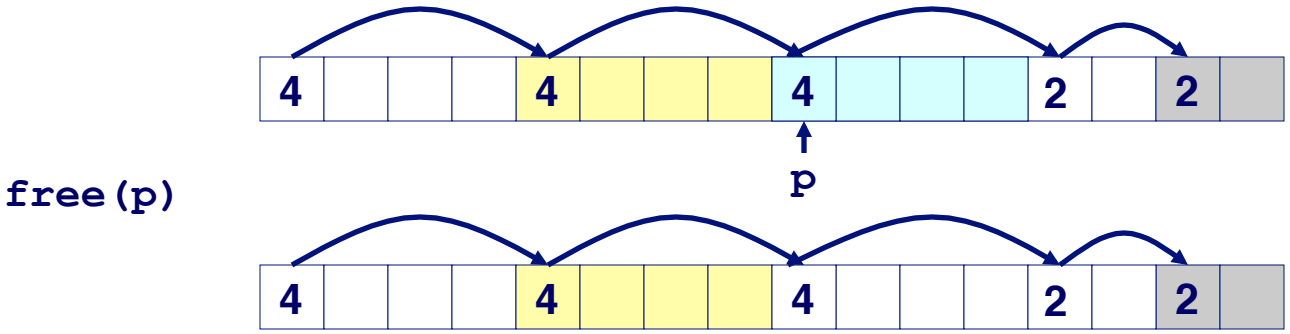
Implicit List: Freeing a Block

Simplest implementation:

- Only need to clear allocated flag

```
void free_block(ptr p) { *p = *p & -2}
```

- But can lead to "false fragmentation"

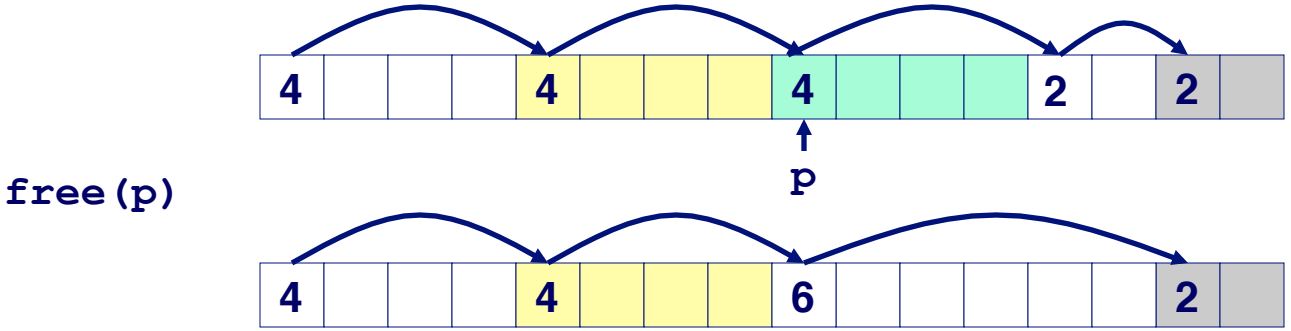


There is enough free space, but the allocator won't be able to find it

Implicit List: Coalescing

Join (*coelesce*) with next and/or previous block if they are free

- Coalescing with next block



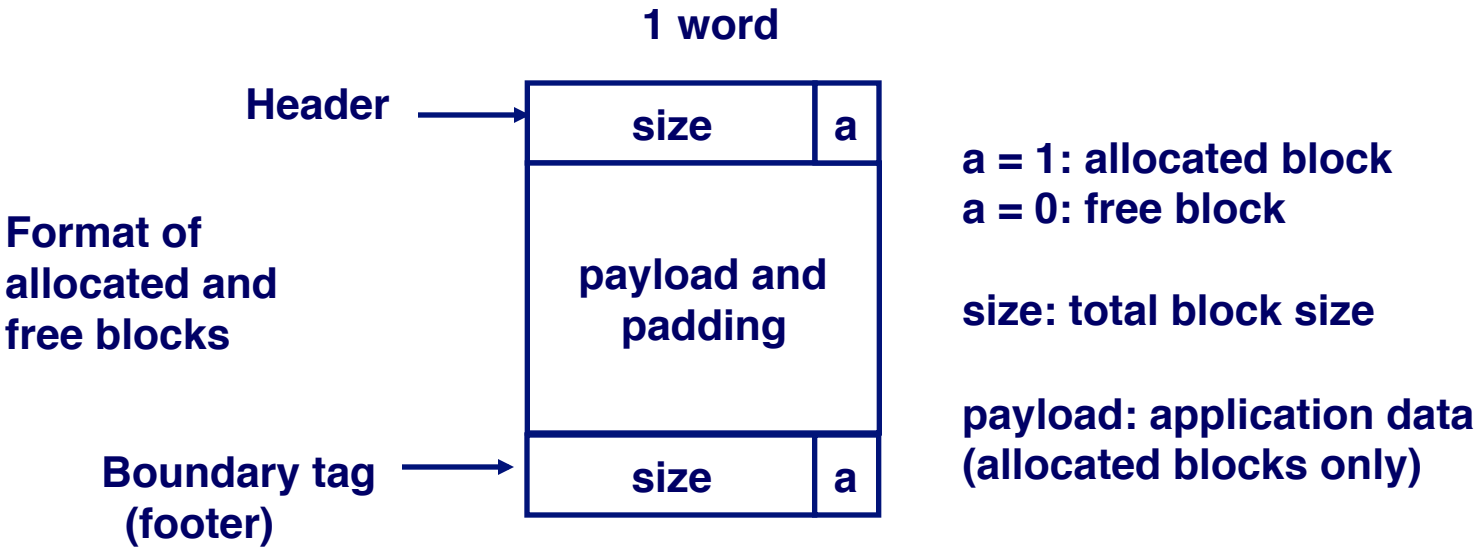

```
void free_block(ptr p) {
    *p = *p & -2;           // clear allocated flag
    next = p + *p;          // find next block
    if ((*next & 1) == 0) // if next block is free...
      *p = *p + *next;    // then add to this block
}
```

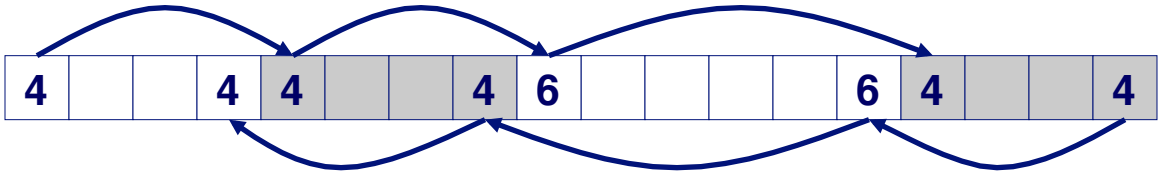
- But how do we coalesce with previous block?

Implicit List: Bidirectional Coalescing

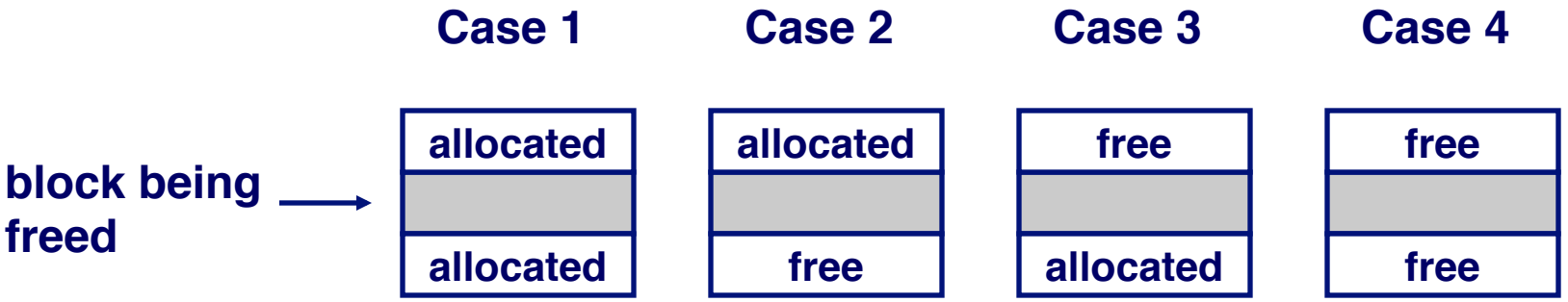
Boundary tags **[Knuth73]**

- Replicate size/allocated word at bottom of free blocks
- Allows us to traverse the "list" backwards, but requires extra space
- Important and general technique!



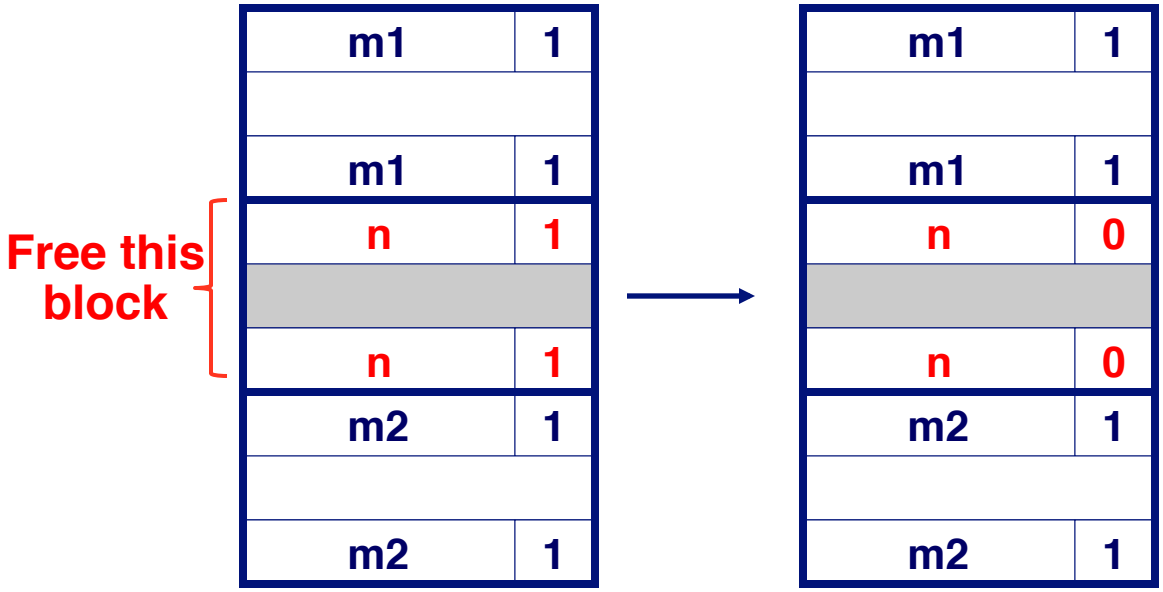


Constant Time Coalescing

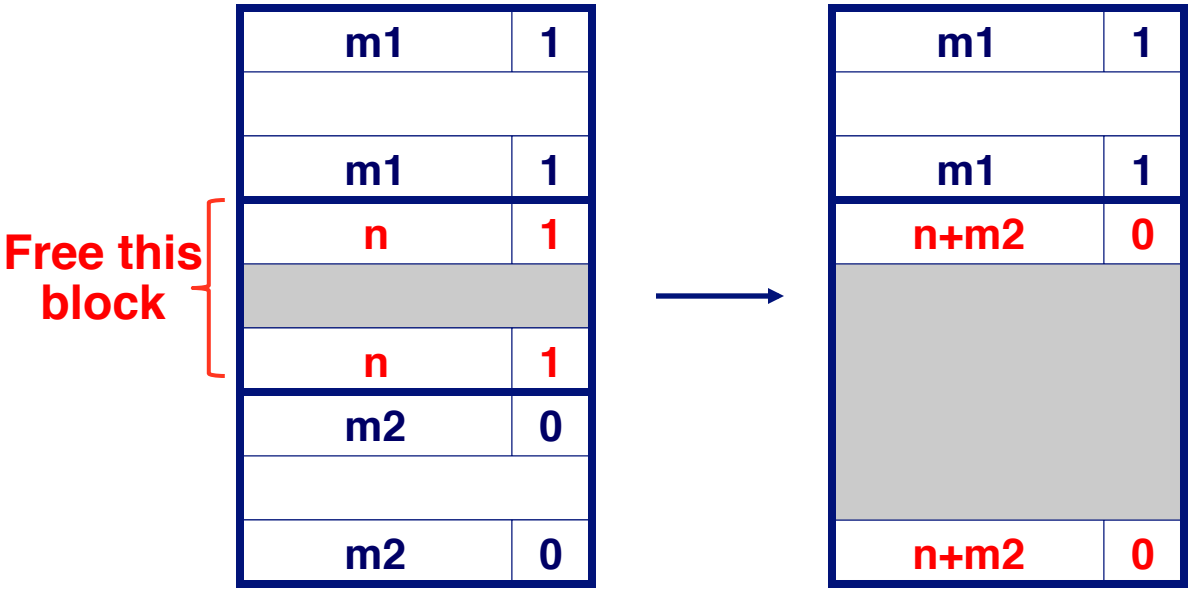


In the worst case 4, only have to fuse three free blocks and update their headers – hence coalescing takes "constant time"

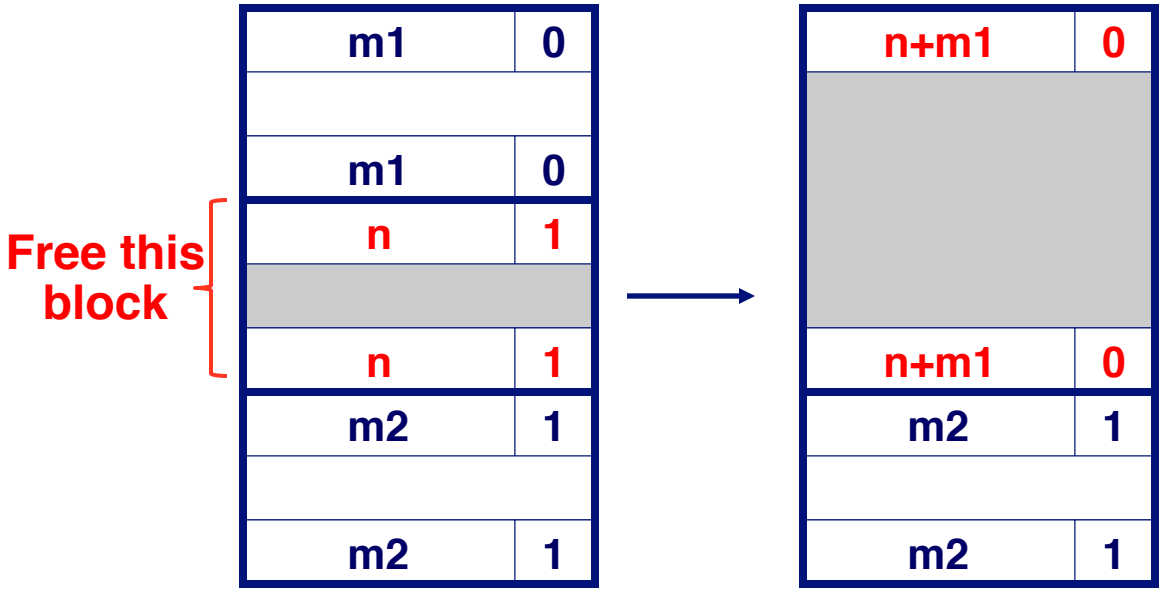
Constant Time Coalescing (Case 1)



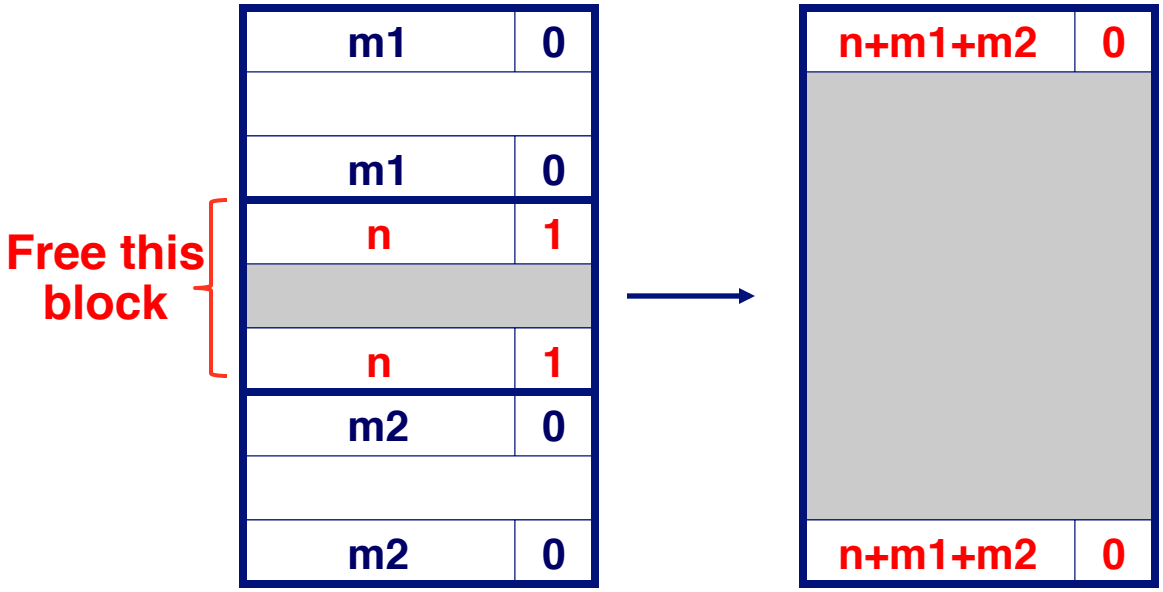
Constant Time Coalescing (Case 2)



Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)



Summary of Key Allocator Policies

Placement policy:

- First fit, next fit, best fit, etc.
- Trades off lower throughput for less fragmentation
  - Interesting observation: segregated free lists (next slides) approximate a best fit placement policy without having to search entire free list.

Splitting policy:

- When do we go ahead and split free blocks?
- How much internal fragmentation are we willing to tolerate?

Coalescing policy:

- Immediate coalescing: coalesce adjacent blocks each time free is called
- Deferred coalescing: try to improve performance of free by deferring coalescing until needed. e.g.,
  - Coalesce as you scan the free list for malloc.
  - Coalesce when the amount of external fragmentation reaches some threshold.

Implicit Lists: Summary

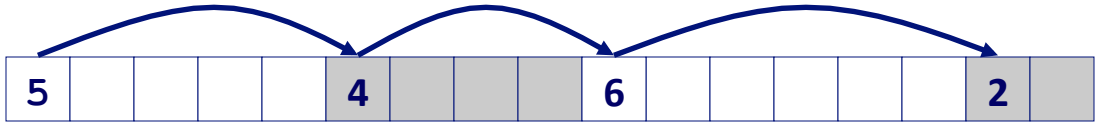
- **Implementation:** very simple
- **Allocate:** linear time worst case
- **Free:** constant time worst case -- even with coalescing
- **Memory usage:** will depend on placement policy
  - First fit, next fit or best fit

Not used in practice for malloc/free because of linear time to allocate. Used in many special purpose applications.

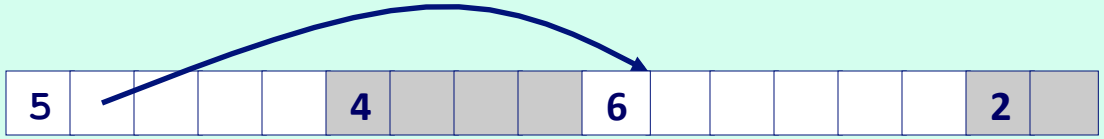
However, the concepts of splitting and boundary tag coalescing are general to *all* allocators.

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



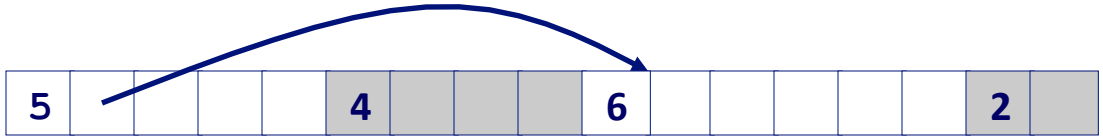
Method 3: *Segregated free list*

- 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

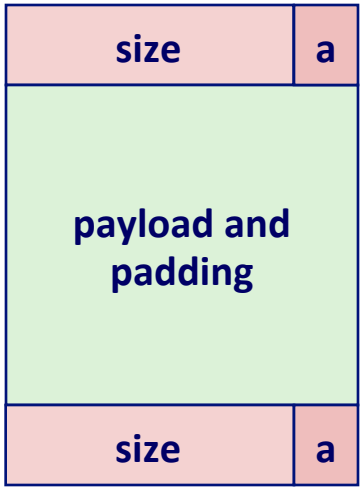
Explicit Free Lists



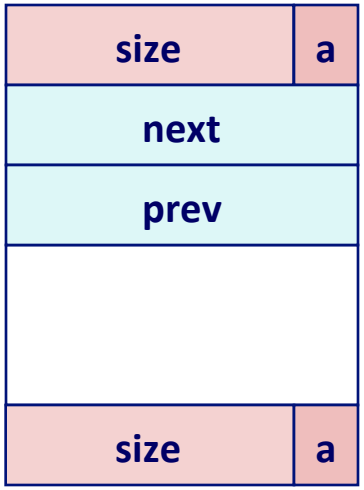
Maintain list(s) of *free* blocks, not *all* blocks

- 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

Allocated (as before)



Free

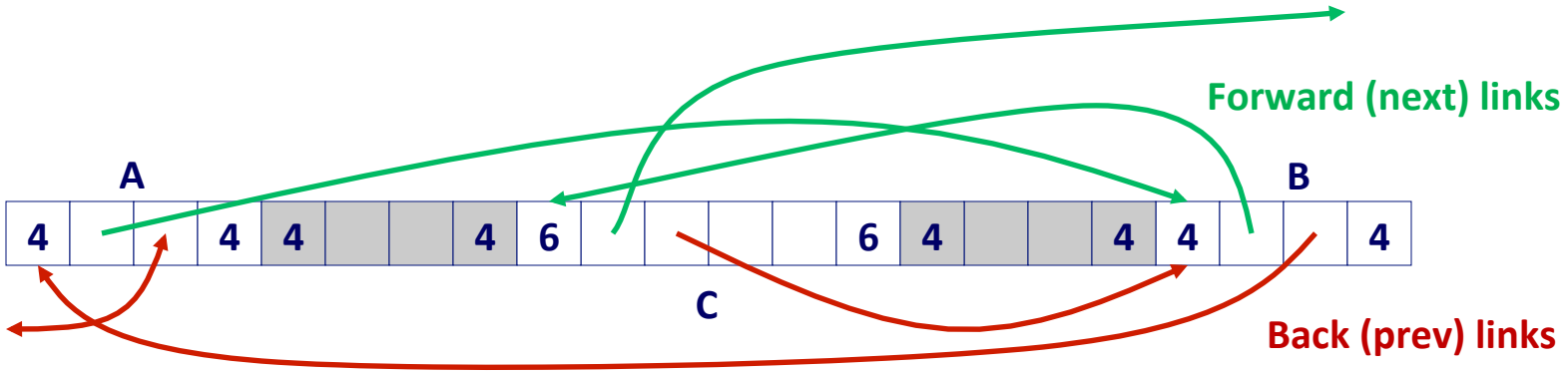


Explicit Free Lists

Logically:

Doubly linked free list gives better performance for malloc'ing than a singly linked free list.

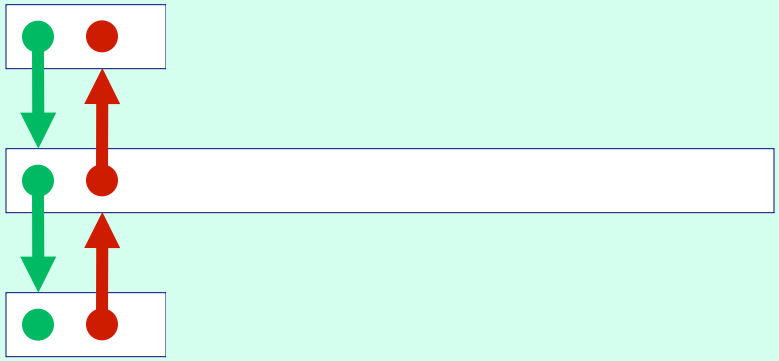
Physically, blocks can be in any order



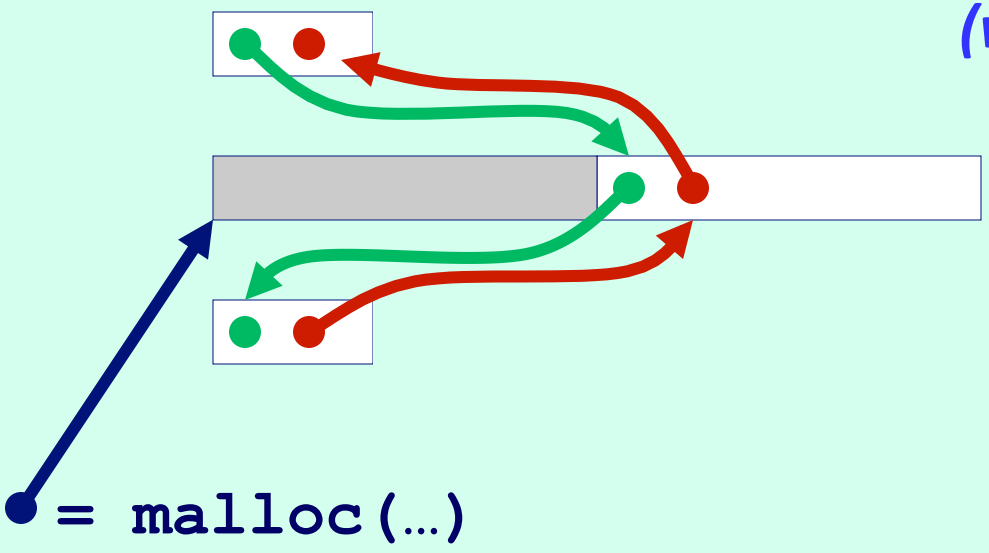
Allocating From Explicit Free Lists

conceptual graphic

Before



After *(with splitting)*



Freeing With Explicit Free Lists

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

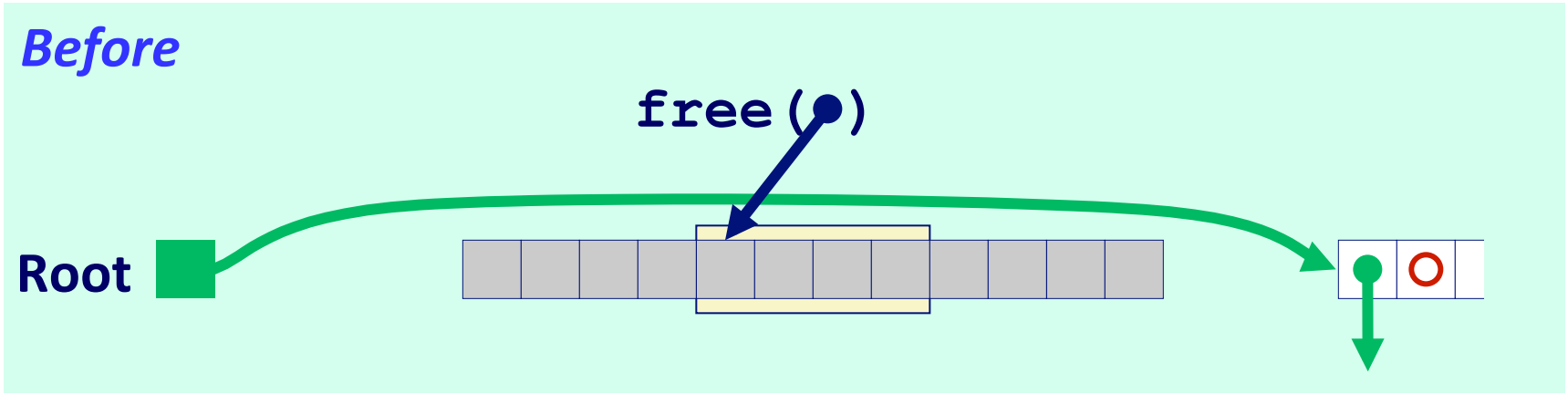
- **LIFO (last-in-first-out) policy**
  - **Insert freed block at the beginning of the free list**
  - ***Pro:* simple and constant time**
  - ***Con:* studies suggest fragmentation is worse than address ordered**

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

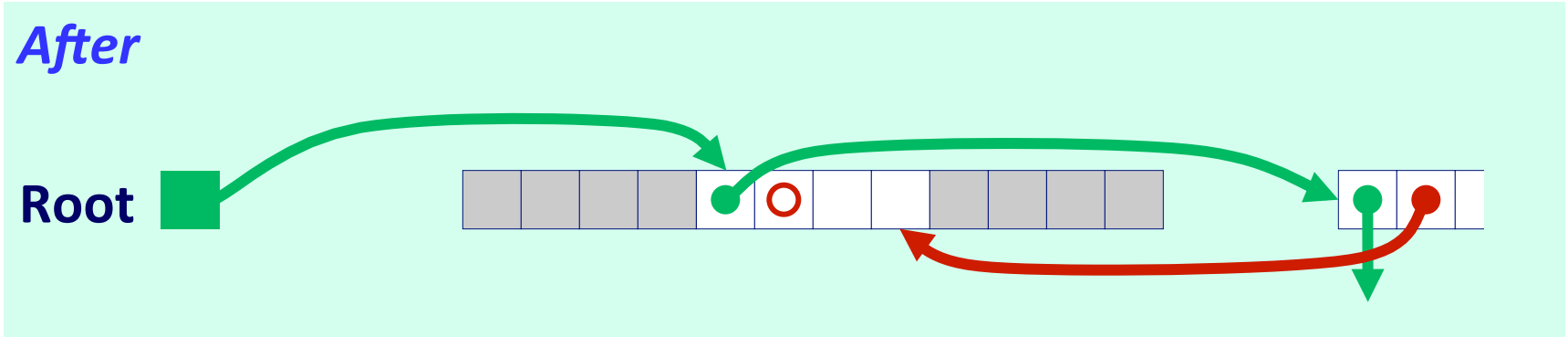
    addr(prev) < addr(curr) < addr(next)
  - ***Con:* requires search**
  - ***Pro:* studies suggest fragmentation is lower than LIFO**

Freeing With a LIFO Policy (Case 1)

conceptual graphic

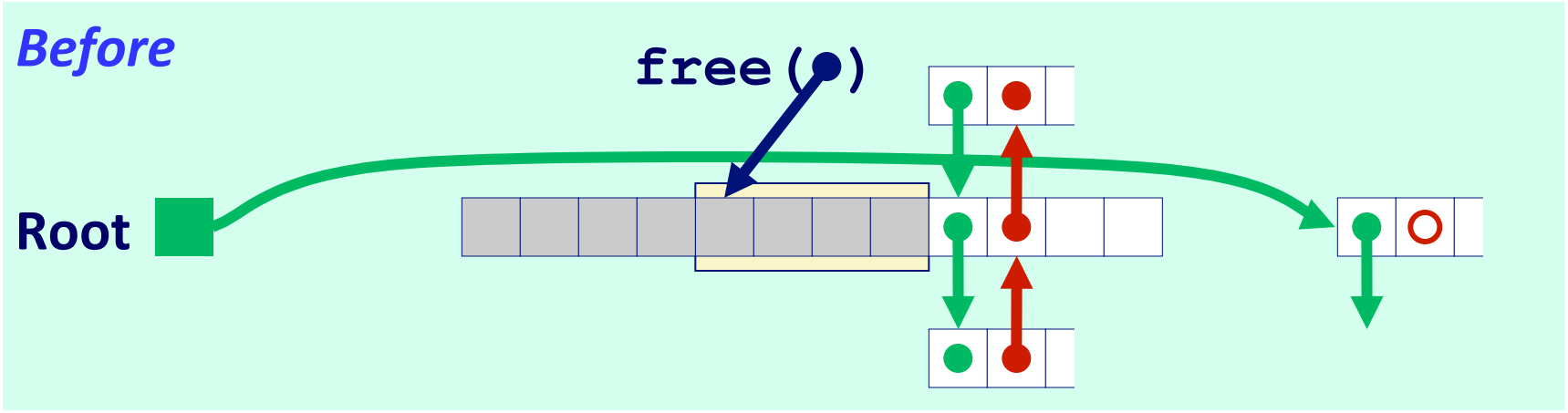


Insert the freed block at the root of the list

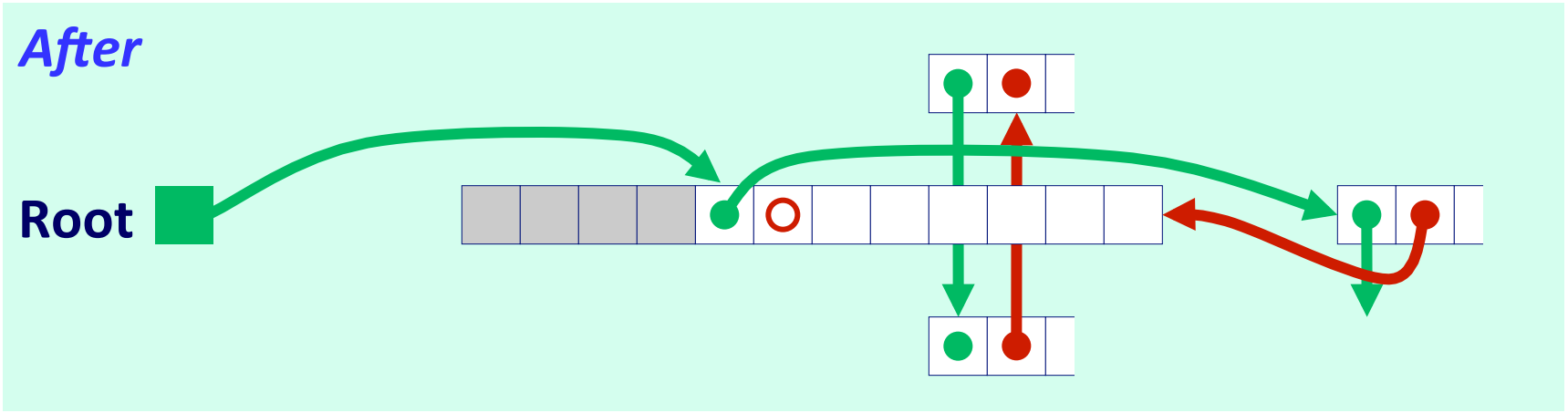


Freeing With a LIFO Policy (Case 2)

conceptual graphic

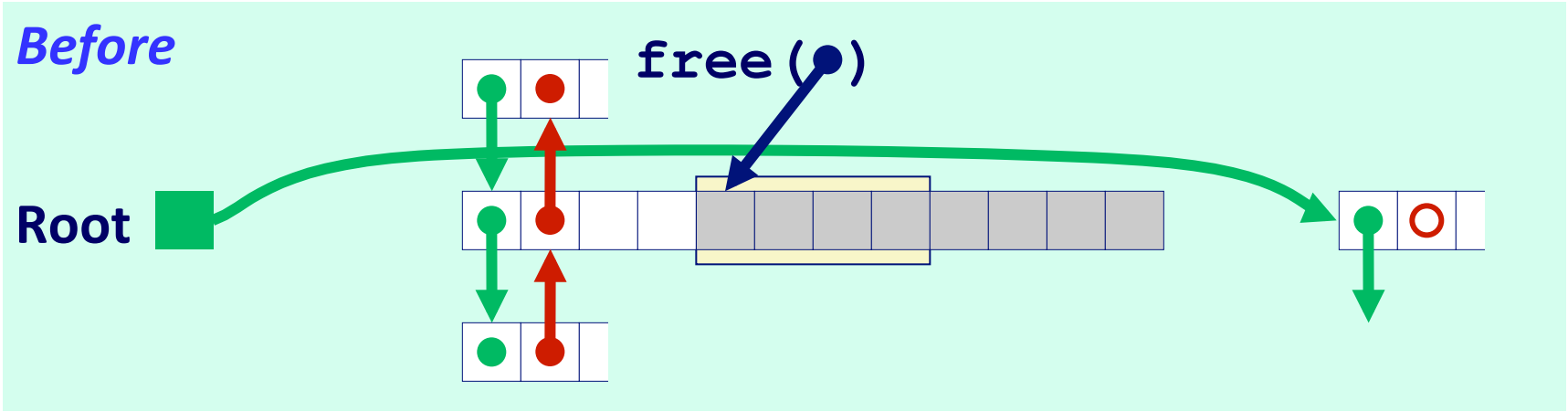


Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list

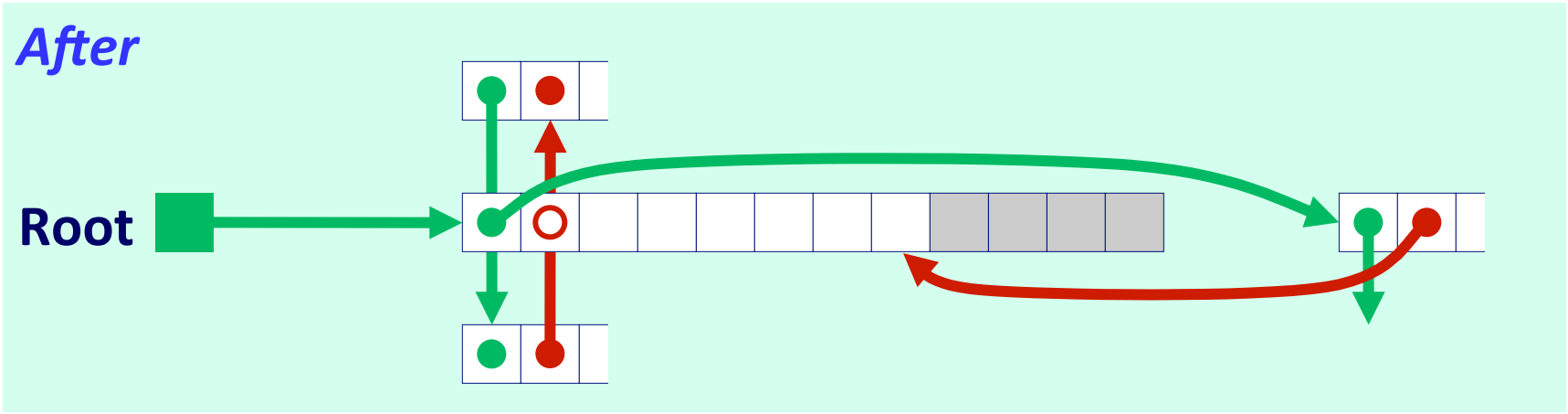


Freeing With a LIFO Policy (Case 3)

conceptual graphic

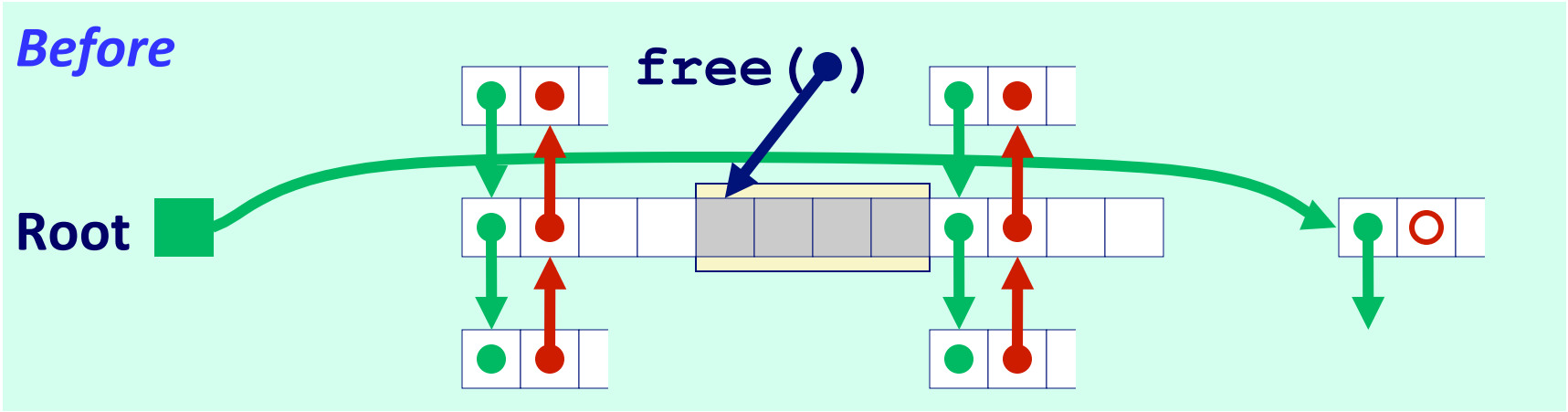


Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list

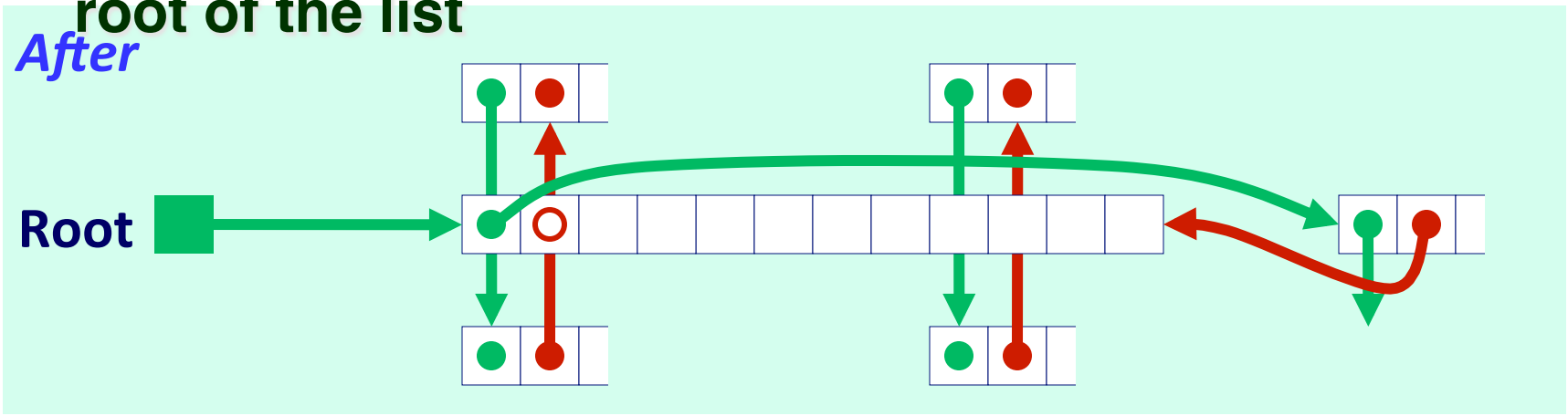


Freeing With a LIFO Policy (Case 4)

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:

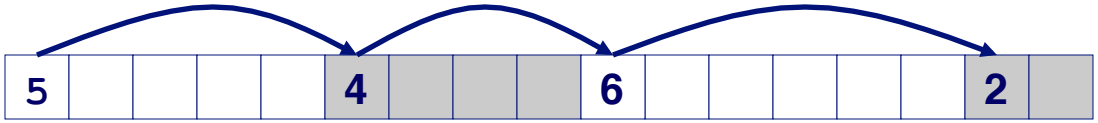
- Allocate is linear time in number of *free* blocks instead of *all* blocks
  - *Much faster* when most of the memory is full
- Slightly more complicated allocate and free since needs to splice blocks in and out of the list
- Some extra space for the links (2 extra words needed for each block)
  - Does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

Most common use of linked lists is in conjunction with segregated free lists

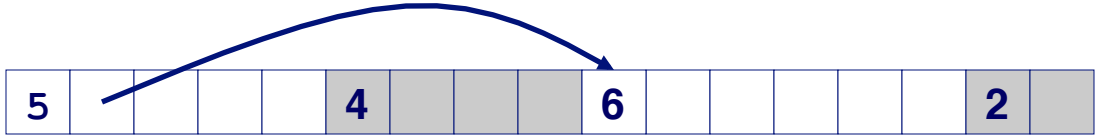
- Keep multiple linked lists of different size classes, or possibly for different types of objects

Keeping Track of Free Blocks

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



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



Method 3: ***Segregated free list***

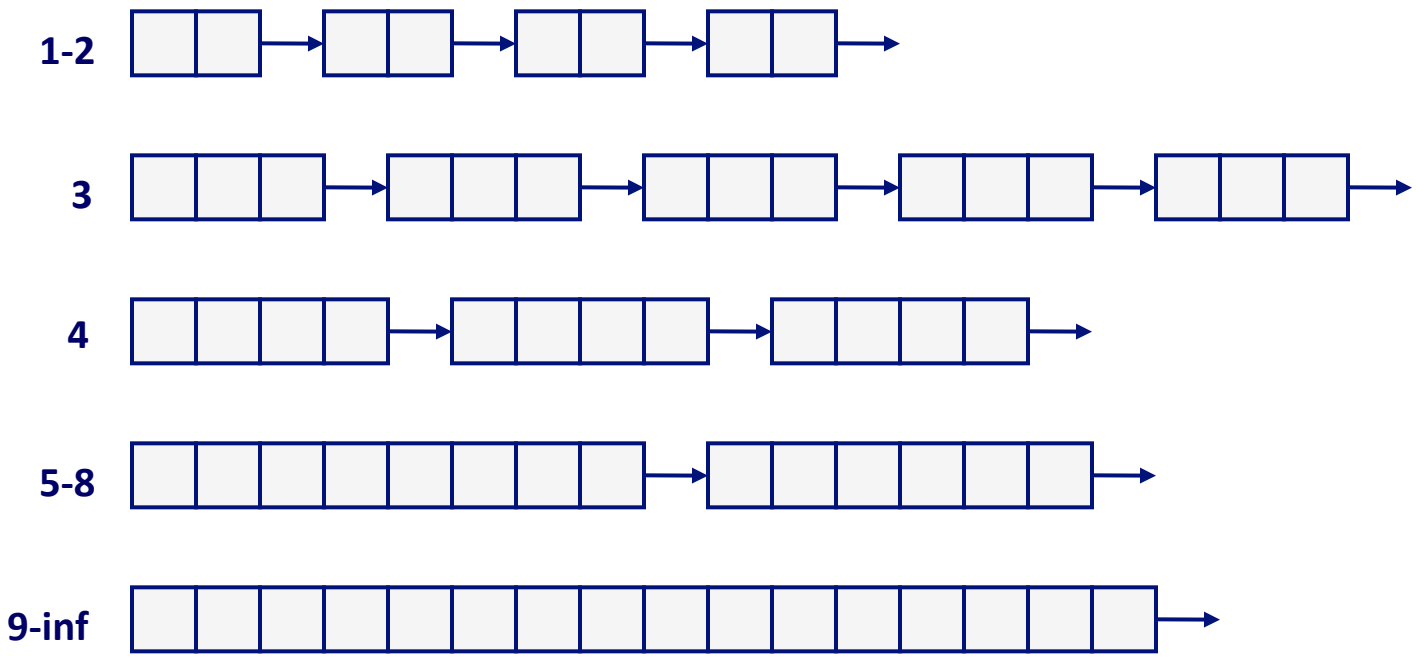
- Different free lists for different size classes

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 (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:

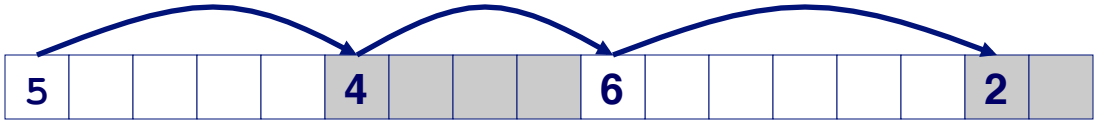
- Coalesce and place on appropriate list (optional)

Advantages of seglist allocators

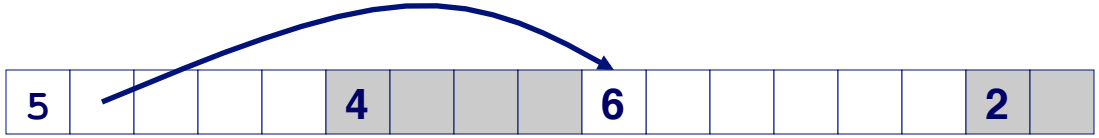
- Higher throughput
  - log time for power-of-two size classes
- Better memory utilization
  - First-fit search of segregated free list approximates a best-fit search of entire heap.
  - Extreme case: Giving each block its own size class is equivalent to best-fit.

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

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,**

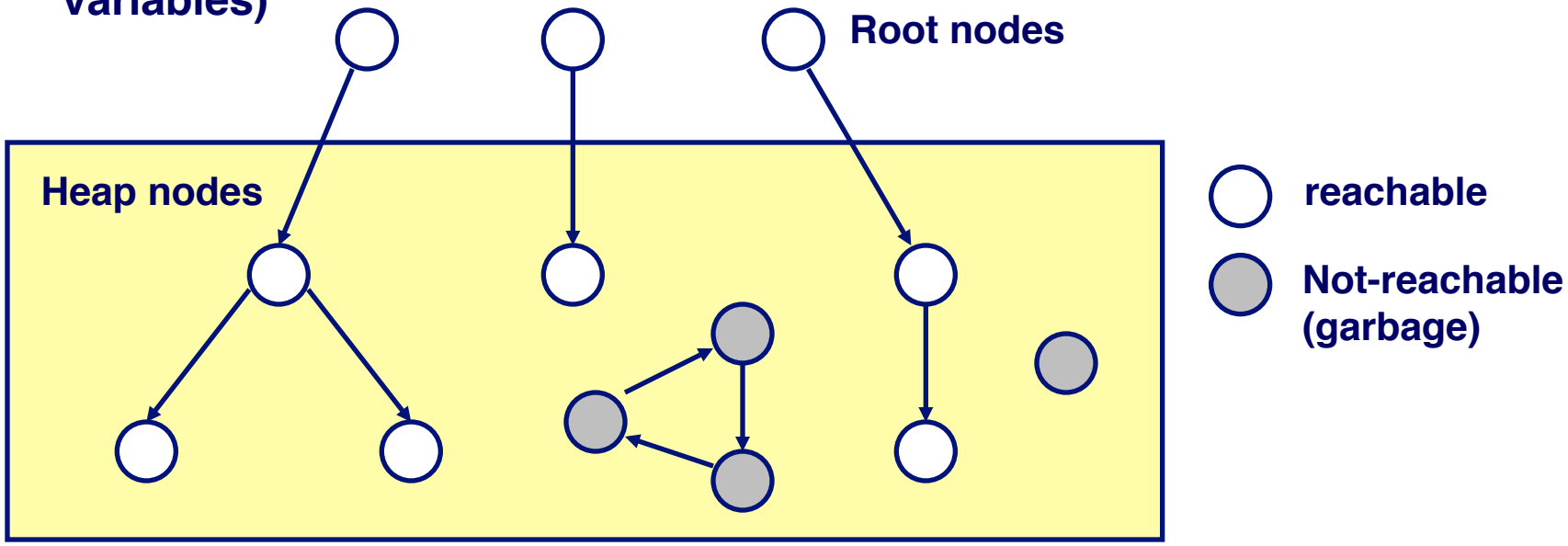
Variants (conservative garbage collectors) exist for C and C++

- **Cannot collect all garbage**

Garbage Collector Finds Unreachable Memory

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)

Mark and Sweep is one type of garbage collection algorithm

Heap-Related Memory Bugs

Calling free(p) with an incorrect pointer p

Freeing blocks multiple times

Referencing freed blocks

Failing to free blocks = Memory Leak!

- heap fills up with allocated but unused memory blocks
- hard to find

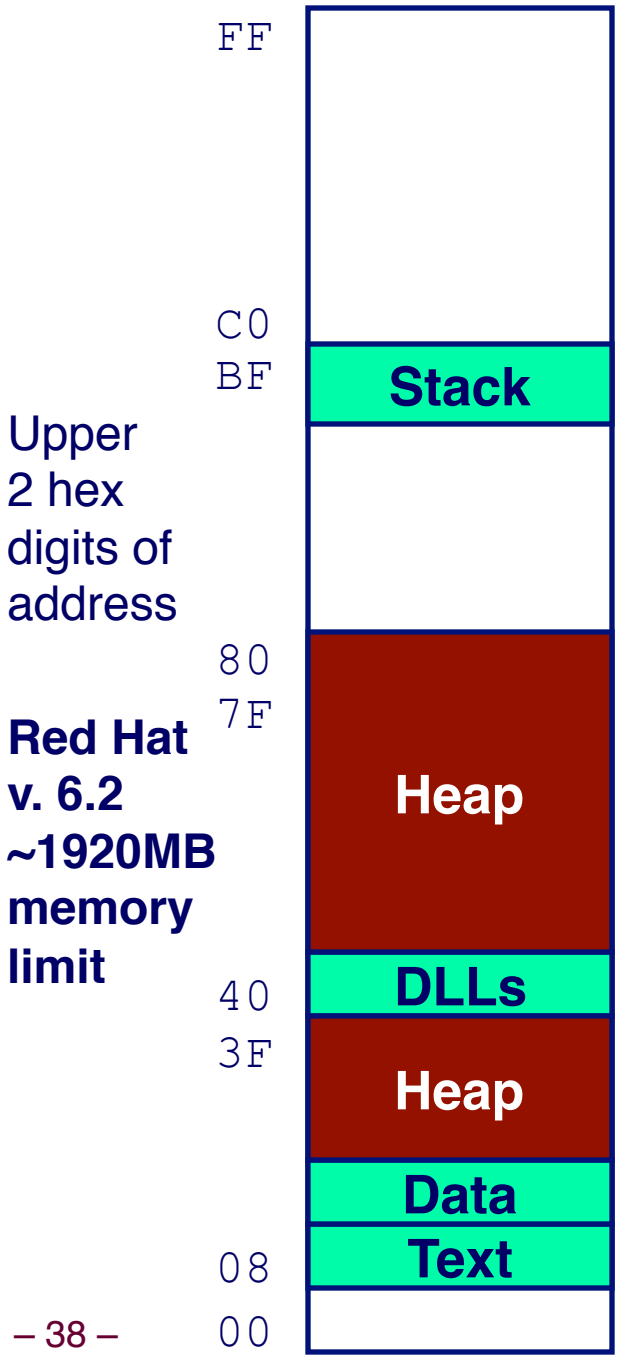
```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   return;
}
```

Use special software tools to find memory leaks

- valgrind, dmalloc, purify, etc.

Supplementary Slides

Linux Memory Layout



Stack

- Runtime stack (8MB limit)

Heap

- Dynamically allocated storage
- When call `malloc`, `calloc`, `new`

DLLs

- Dynamically Linked Libraries
- Library routines (e.g., `printf`, `malloc`)
- Linked into object code when first executed

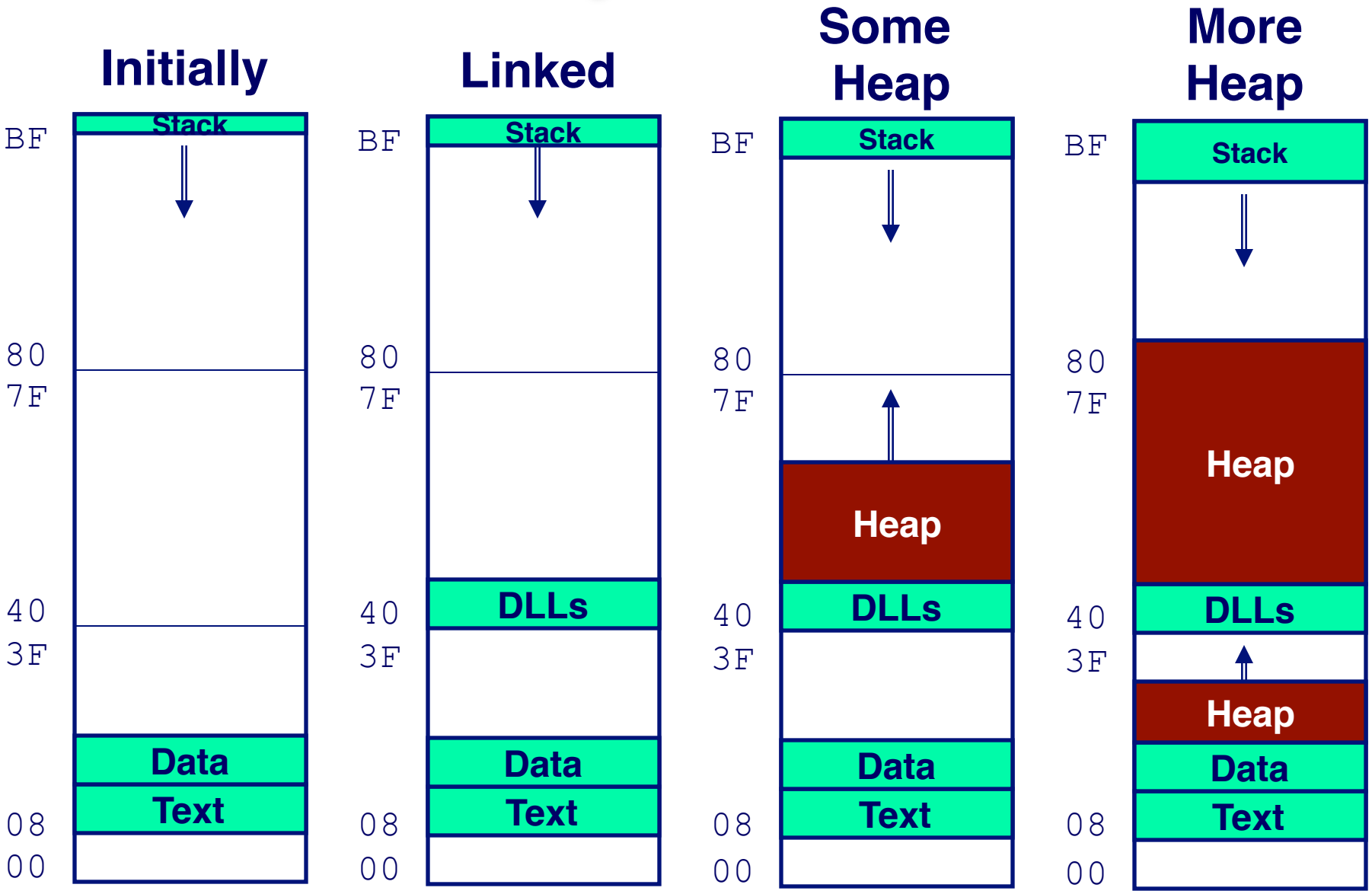
Data

- Statically allocated data
- E.g., arrays & strings declared in code

Text

- Executable machine instructions
- Read-only

Linux Memory Allocation



Constraints

Applications:

- Can issue arbitrary sequence of allocation and free requests
- Free requests must correspond to an allocated block

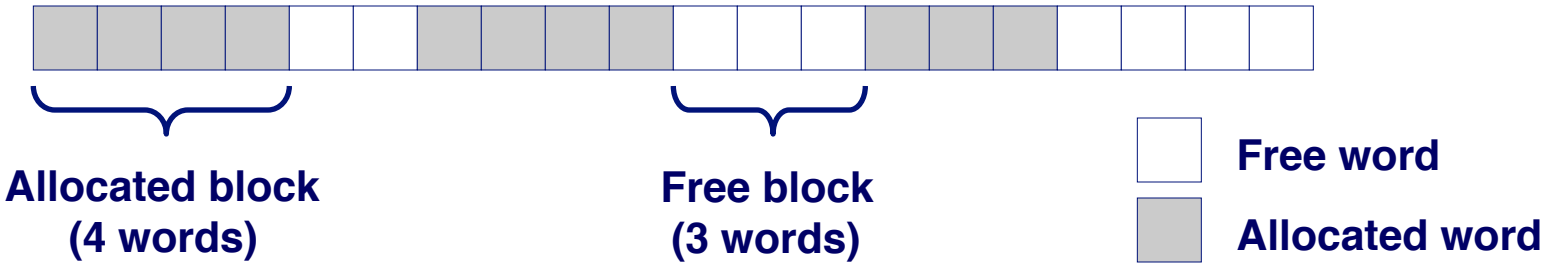
Allocators

- Can't control number or size of allocated blocks
- Must respond immediately to all allocation requests
  - *i.e.*, can't reorder or buffer requests
- Must allocate blocks from free memory
  - *i.e.*, can only place allocated blocks in free memory
- Can't move the allocated blocks once they are allocated
  - *i.e.*, compaction is not allowed
- Must align blocks so they satisfy all alignment requirements
  - 8 byte alignment for GNU malloc (`libc` malloc) on Linux boxes
- Can only manipulate and modify free memory

Assumptions

Model the Heap memory as a sequence of words

- Memory is word addressed (each word can hold a pointer)



Performance Goals: Throughput

Given some arbitrary sequence of malloc and free requests:

- $R_0, R_1, ..., R_k, ..., R_{n-1}$

Want to maximize throughput and peak memory utilization.

- **These goals are often conflicting**

Throughput:

- **Number of completed requests per unit time**
- **Example:**
  - **5,000 malloc calls and 5,000 free calls in 10 seconds**
  - **Throughput is 1,000 operations/second.**

Performance Goals: Peak Memory Utilization

Given some sequence of malloc and free requests:

- $R_0, R_1, ..., R_k, ..., R_{n-1}$

Def: Aggregate payload P_k:

- `malloc(p)` **results in a block with a *payload* of** `p` **bytes..**
- **After request R_k has completed, the *aggregate payload P_k* is the sum of currently allocated payloads.**

Def: Current heap size is denoted by H_k

- **Assume that H_k is monotonically nondecreasing**

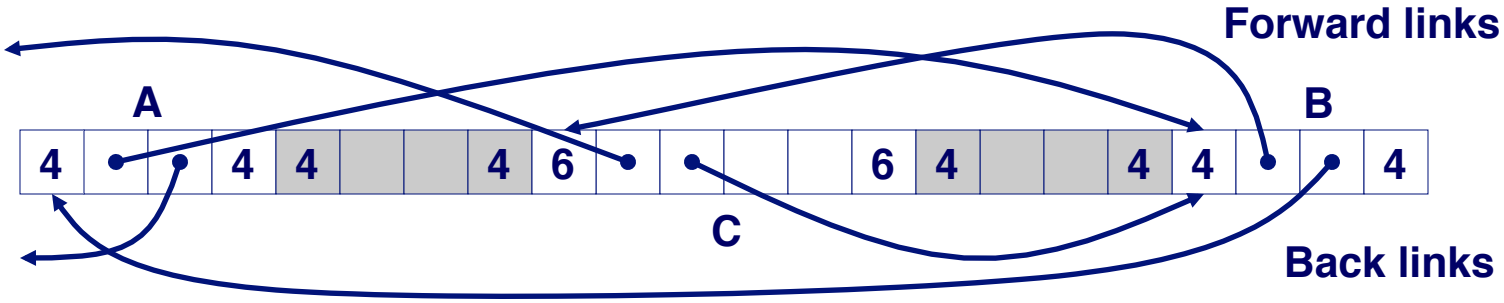
Def: Peak memory utilization:

- **After *k* requests, *peak memory utilization* is:**
  - $U_k = (max_{i<k} P_i) \ / \ H_k$

Explicit Free Lists

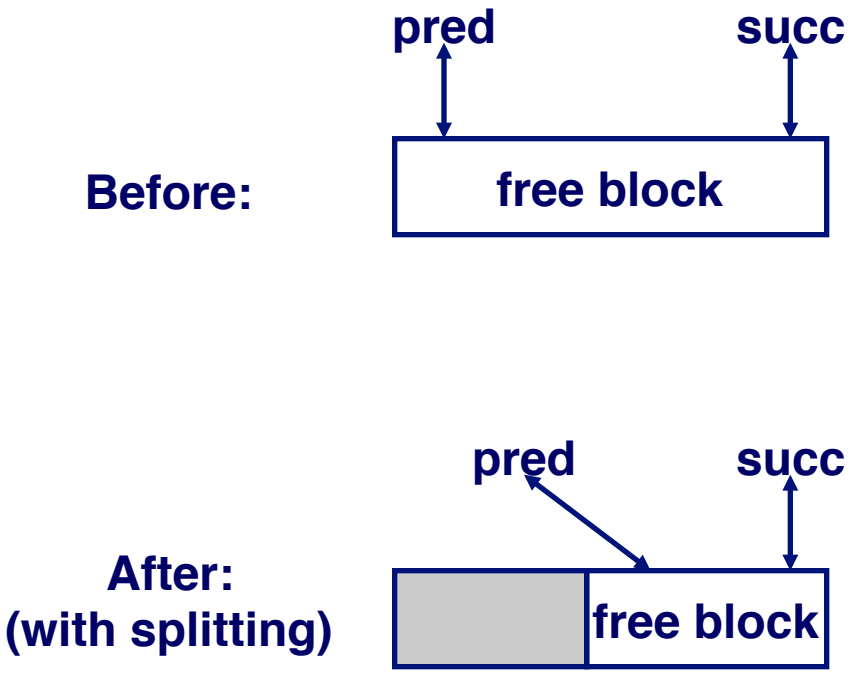
Use data space for link pointers

- 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



Freeing With Explicit Free Lists

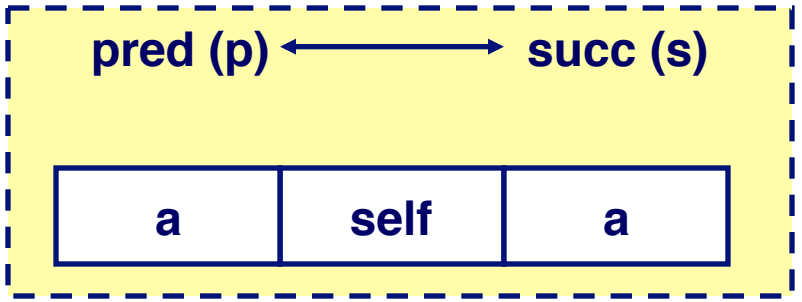
***Insertion policy*: Where in the free list do you put a newly freed block?**

- 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. addr(pred) < addr(curr) < addr(succ)
  - Con: requires search
  - Pro: studies suggest fragmentation is better than LIFO

Freeing With a LIFO Policy

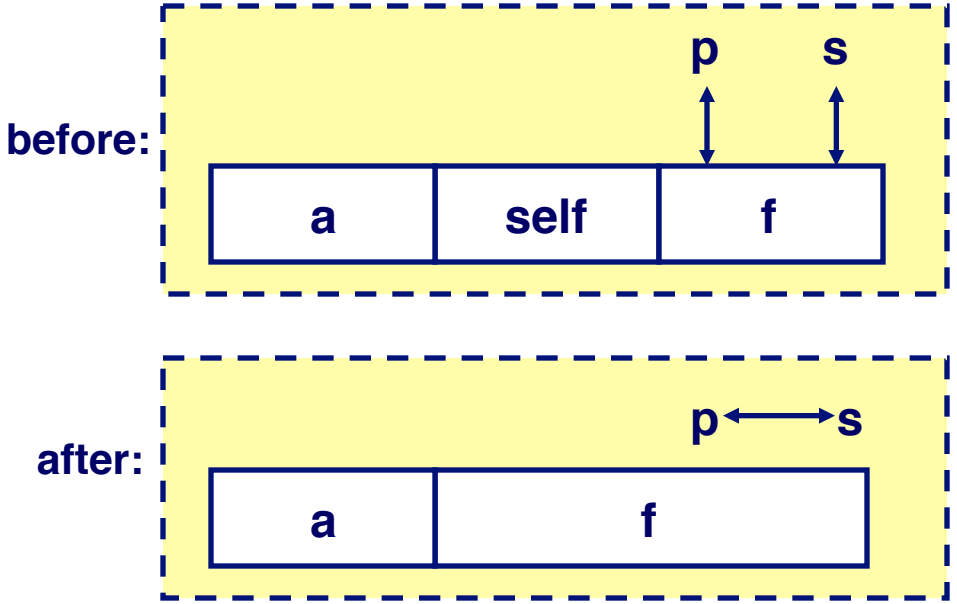
Case 1: a-a-a

- Insert self at beginning of free list



Case 2: a-a-f

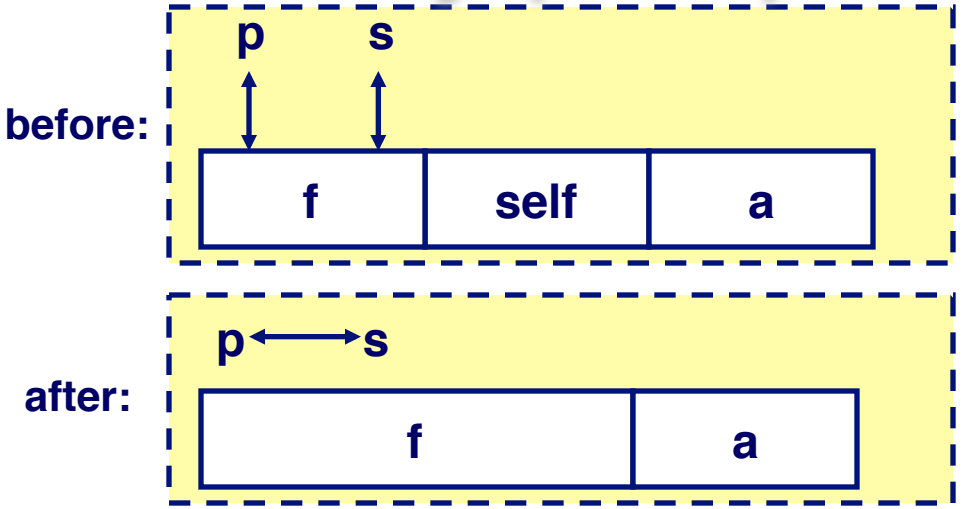
- Splice out next, coalesce self and next, and add to beginning of free list



Freeing With a LIFO Policy (cont)

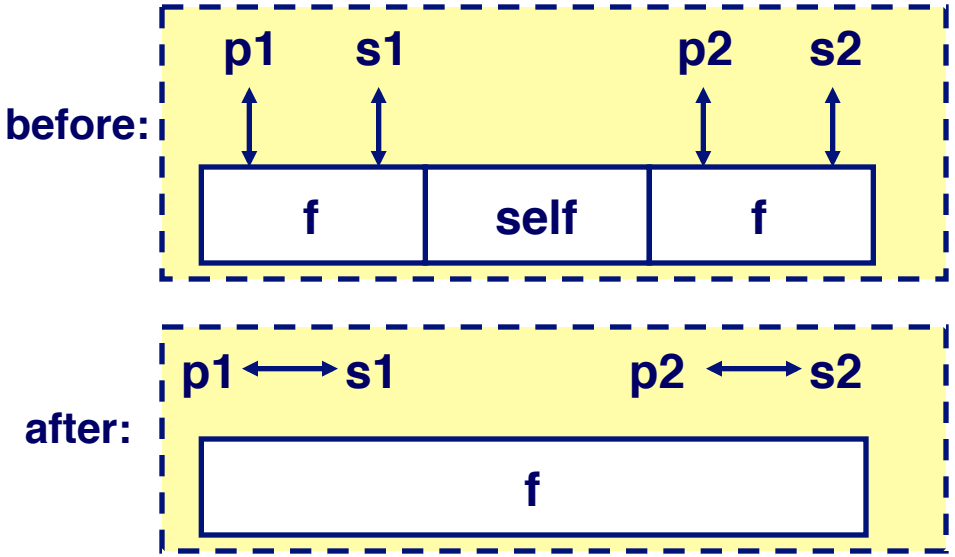
Case 3: f-a-a

- Splice out prev, coalesce with self, and add to beginning of free list



Case 4: f-a-f

- Splice out prev and next, coalesce with self, and add to beginning of list



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)

Segregated Fits

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

To free a block:

- Coalesce and place on appropriate list (optional)

Tradeoffs

- Faster search than sequential fits (i.e., log time for power of two size classes)
- Controls fragmentation of simple segregated storage
- Coalescing can increase search times
  - Deferred coalescing can help

Simple Segregated Storage

Separate heap and free list for each size class

No splitting

To allocate a block of size n:

- If free list for size n is not empty,
  - allocate first block on list (note, list can be implicit or explicit)
- If free list is empty,
  - get a new page
  - create new free list from all blocks in page
  - allocate first block on list
- Constant time

To free a block:

- Add to free list
- If page is empty, return the page for use by another size (optional)

Tradeoffs:

- Fast, but can fragment badly

Harsh Reality

Memory Matters

Memory is not unbounded

- It must be allocated and managed
- Many applications are memory dominated
  - Especially those based on complex, graph algorithms

Memory referencing bugs especially pernicious

- Effects are distant in both time and space

Memory performance is not uniform

- Cache and virtual memory effects can greatly affect program performance
- Adapting program to characteristics of memory system can lead to major speed improvements