Memory Allocation I

CSE 351 Summer 2024

Instructor:
Ellis Haker

Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang

Administrivia

- Today:
  - **Lab 5 released**
    - Due Wednesday, 1/14
- Monday, 1/5
  - RD21 due (1pm)
  - HW19 due (11:59pm)
- Wednesday, 1/7
  - RD22 due (1pm)
  - HW20 due (11:59pm)
  - **Lab4 due (11:59pm)**

Dynamic Memory Allocation

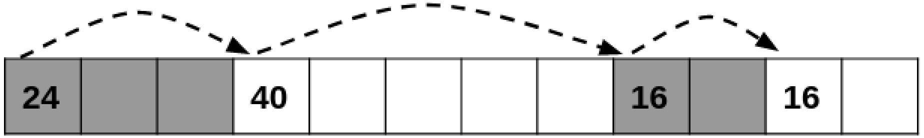
- Overview
  - Introduction and goals
  - Allocation and deallocation (free)
- Malloc and free in C
  - Common memory-related bugs in C
- Fragmentation
- **Explicit allocation implementation**
  - **Implicit free lists**
  - **Splitting and coalescing**
  - Explicit free lists (Lab 5)
- Implicit deallocation: garbage collection

When you allocate variables on the stack

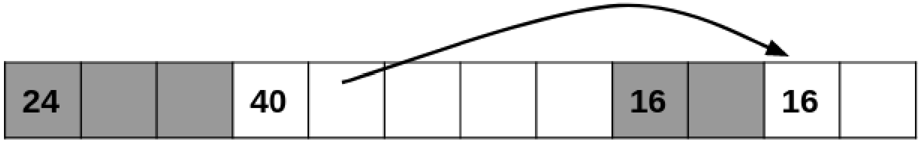
When you allocate variables on the heap

Recap: Keeping track of free blocks

- **Implicit Free List**
  - Use pointer arithmetic to traverse through entire heap until we find a free bock



- **Explicit Free List**
  - Free block stores pointer to the next free block, forming a linked list

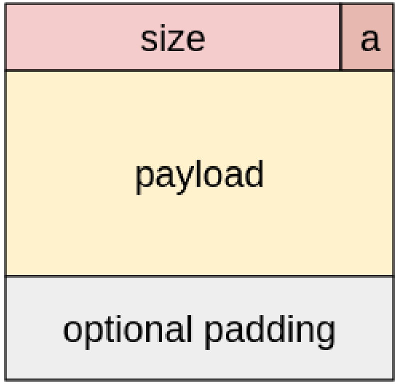


- Others not covered in this class
  - **Segregated free lists** (different lists for each object type), sorting blocks by size

Recap: Implicit Free Lists

- For each block, we need to store: **`size`**, **`is_allocated`**
  - Could use two works, but kinda wasteful…
- Recap: if size is a multiple of 2^n, then lowest n bits of the size are always 0
  - Use the lowest bit of header word to store **`is_allocated`** flag
  - When reading **`size`**, mask this bit out

Block format:



a = 1 if allocated, 0 if free

size = total block size in bytes

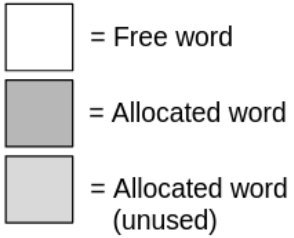
payload for allocated blocks only

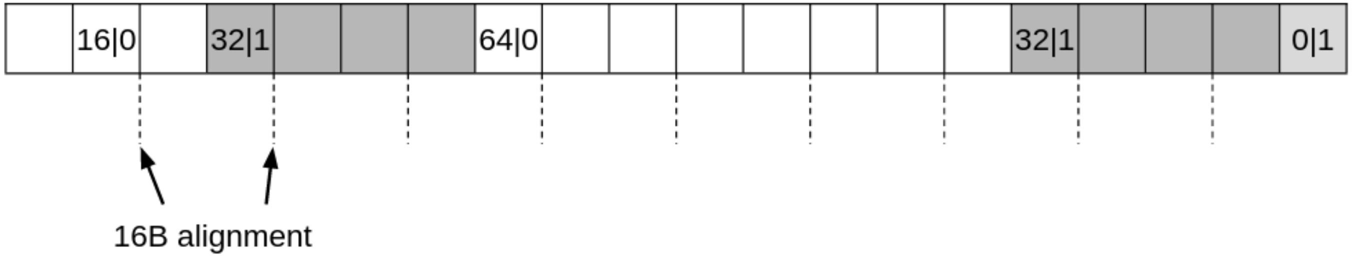
If the header value is h:

```
h = size | a
a = h & 1
size = h & ~1
```

Implicit Free List Example

- Each block begins with a header containing size and the allocated flag
- Payload is 16B aligned
  - May require padding
- Extra “header” to mark the end of the heap: 0|1
  - Adds 8B of *external* fragmentation
- Sequence of blocks in heap (size|alloc): 16|0, 32|1, 64|0, 32|1





Implicit Free List: Finding a Free Block

- **First fit**: start from beginning, choose first free block that fits

```
p = heap_start;
while ((p < end) &&                       // while not past the end of heap
        ((*p & 1) || (*p <= len))) {     // while p allocated or too small
    int size = p & ~1;
    p += p, size;                         // Go to next block, (UNSCALED +)
}  // p points to selected block or end
```

  - Can take linear time in total number of blocks
  - Can cause “splinters” at beginning of the heap

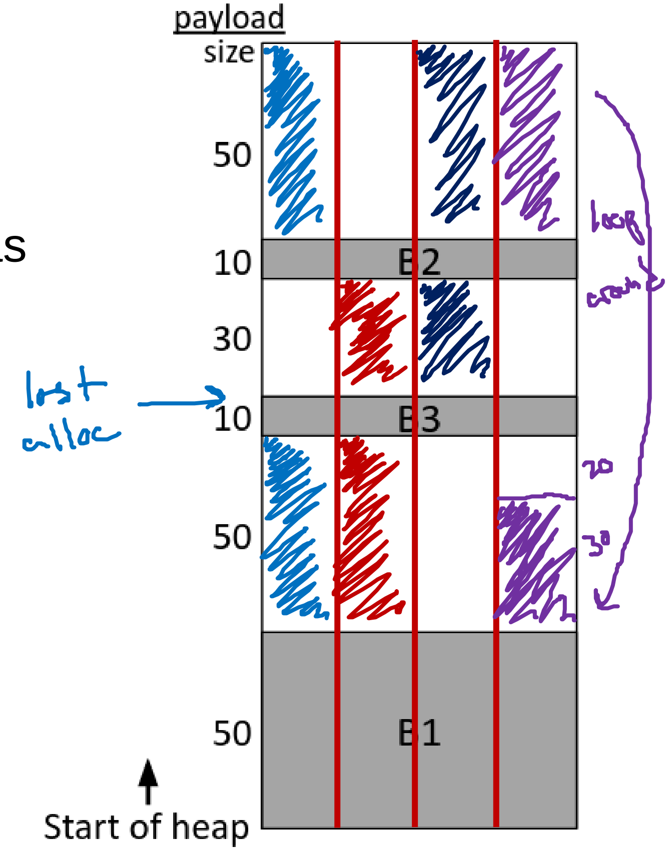
this is pseudocode
ignoring the type of p and pointer arithmetic scaling

Implicit Free List: Finding a Free Block (pt 2)

- **Next fit**: like first fit, but **search list starting from where previous search finished**
  - Often faster than first-fit, avoid scanning through as many allocated blocks
  - Some research suggests fragmentation is worse
- **Best fit**: search through *all* free blocks, choose the one that's large enough with **fewest bytes left over**
  - Usually helps fragmentation
  - Worse throughput, have to look through all blocks

Polling Question

- Which allocation strategy and requests remove <u>external</u> fragmentation in this Heap? Note: B3 was the last fulfilled request.
   - **A)** Best-fit:
      - malloc(50), malloc(50) 30B
   - **B)** First-fit:
      - malloc(50), malloc(30) 0B (end of heap doesn't count)
   - **C)** Next-fit:
      - malloc(30), malloc(50) 50B
   - **D)** Next-fit:
      - malloc(50), malloc(30) 50B



Allocating a Free Block

- Easy with implicit free list, just set the allocated bit
- What if the block we choose is much larger than requested?
    - **Split** into two blocks

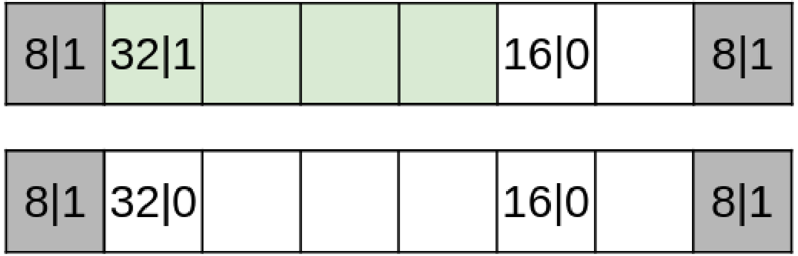
```
void split(ptr b, int bytes) {               // bytes = desired block size
  int newsize = ((bytes+15) >> 4) << 4;   // round up to multiple of 16
  int oldsize = *b;                        // Why not mask out low bit?
  *b = newsize;                            // initially unallocated
  if (newsize < oldsize)
    *(b+newsize) = oldsize - newsize;     // Set length in remaining
}                                          // part of block (UNSCALED +)
```

Ex: `malloc(16)`

Freeing a Block

- Simplest implementation, just set allocated bit to false
  - This can lead to “false fragmentation”

<u>Ex</u>: `free(p)`



What happens if we call `malloc(40)`? *Can’t find a free block!*

- Solution: **coalesce** adjacent free blocks

Coalescing with Next block

```
void free(ptr p) {               // p points to payload
    ptr b = p - WORD;            // b points to block header (UNSCALED -)
    *b &= ~1;                    // clear allocated bit
    ptr next = b + *b;           // find next block (UNSCALED +)
    if ((*next & 1) == 0)        // if next block is not allocated,
      *b += *next;               // add its size to this block
}
```

8|1	32|0				16|0		8|1

→

8|1	48|0				16|0		8|1

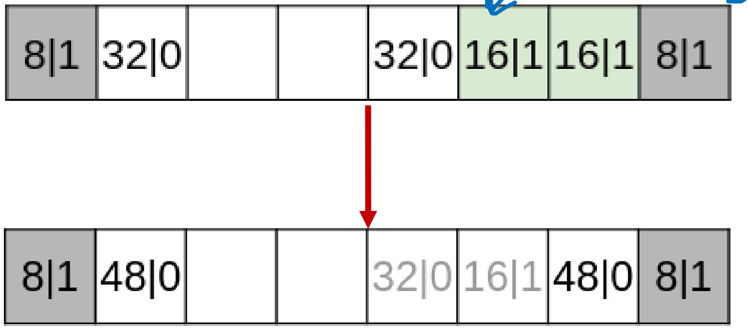
old header is still here, we just ignore it

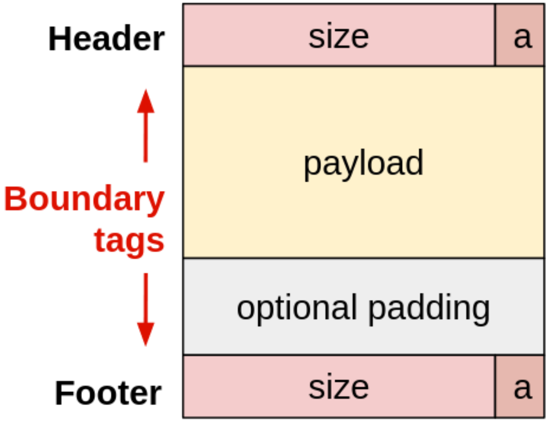
- How do we coalesce with the *preceding* block, though?

Coalescing with Previous Block

- Keep a **footer** for each block
  - Copy of the header at the end of a block
  - Header + footer = **boundary tags**
- When coalescing, check footer immediately before current block in memory
  - If free, coalesce blocks

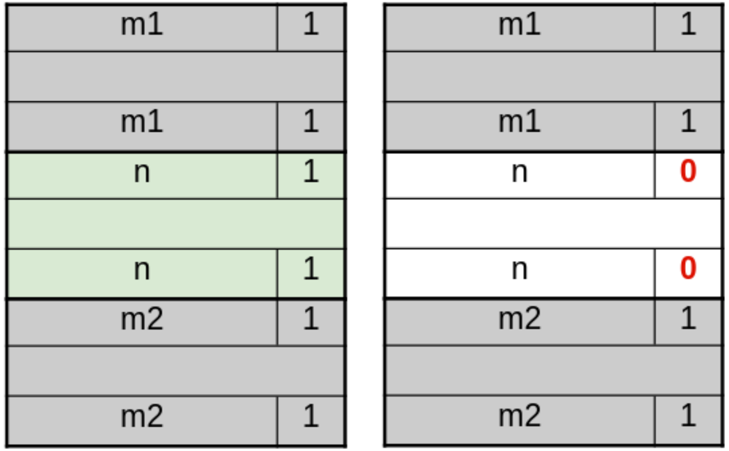
blocks we're freeing



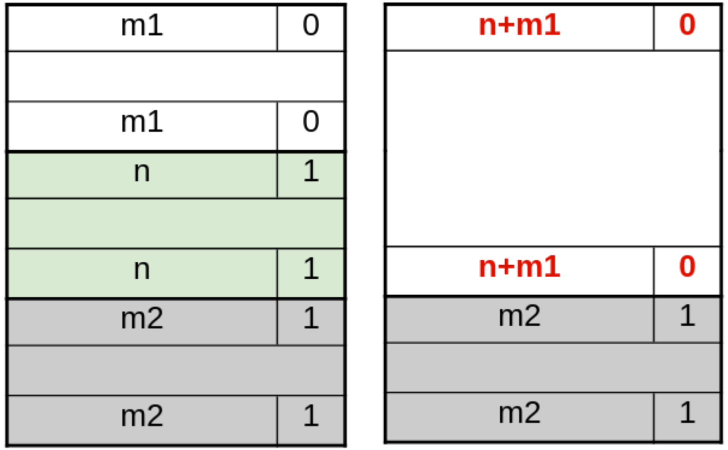


Coalescing

Case 1:



Case 2:



Coalescing (pt 2)

Case 3:

m1	1
m1	1
n	1
n	1
m2	0
m2	0

m1	1
m1	1
n+m2	**0**
n+m2	**0**

Case 4:

m1	0
m1	0
n	1
n	1
m2	0
m2	0

n+m1+m2	**0**
n+m1+m2	**0**

Implicit Free List Review Questions

- When coalescing free blocks, how many neighboring blocks do we need to check on either side? Why? Just one!

  If we always coalesce in free, there will never be 2 adjacent free blocks

- If I want to check the size of the *n*th block forward from the current block, how many memory accesses do I need to make?

  n+1

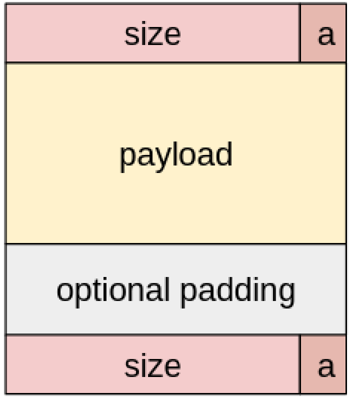
  check current block's size to get pointer to next, keep doing that until you reach n blocks ahead, then check that block's size

Dynamic Memory Allocation

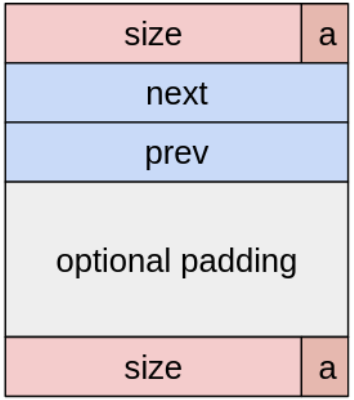
- Overview
  - Introduction and goals
  - Allocation and deallocation (free)
- Malloc and free in C
  - Common memory-related bugs in C
- Fragmentation
- **Explicit allocation implementation**
  - Implicit free lists
  - Splitting and coalescing
  - **Explicit free lists (Lab 5)**
- Implicit deallocation: garbage collection

Explicit Free Lists

Allocated block:
(same as implicit free list)



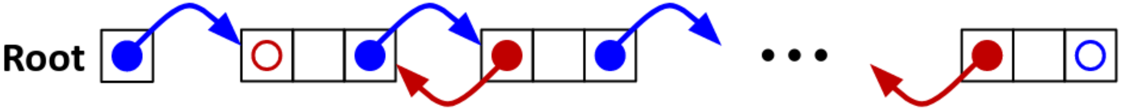
Free block:



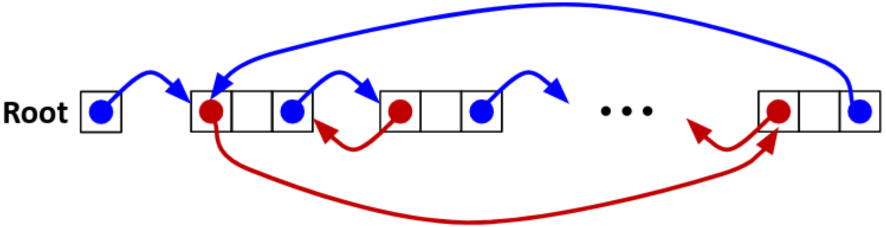
- Create a linked list of free blocks only, rather than having to search through all blocks
    - Since only free blocks are in the list, can use the space that would be the payload for an allocated block
    - Still need boundary tags for coalescing

Doubly-Linked Lists

- **Linear**
  - Root pointer points to first node
  - First node's prev = NULL
  - Last node's next = NULL
  - Better for first-fit, best-fit



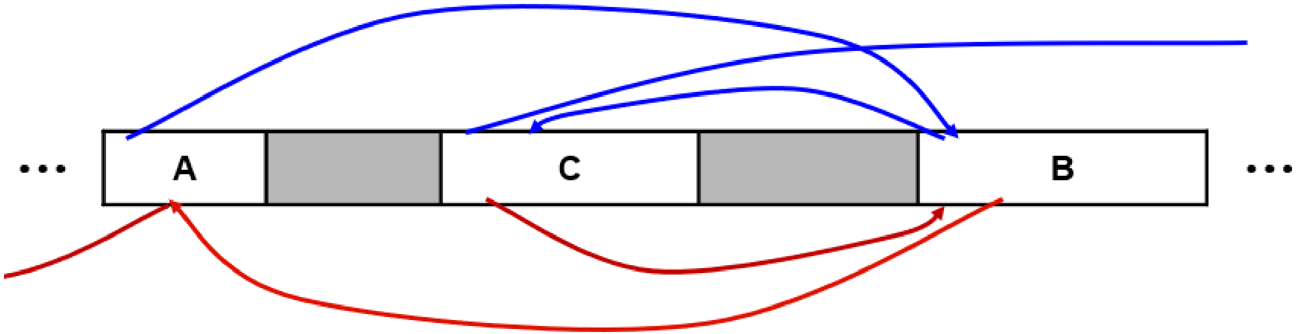
- **Circular**
  - Still need root pointer to tell you when to start
  - First and last node connected (no NULL pointers)
  - Better for next-fit, best-fit



Explicit Free Lists (pt 2)

- **Logically**: doubly-linked list

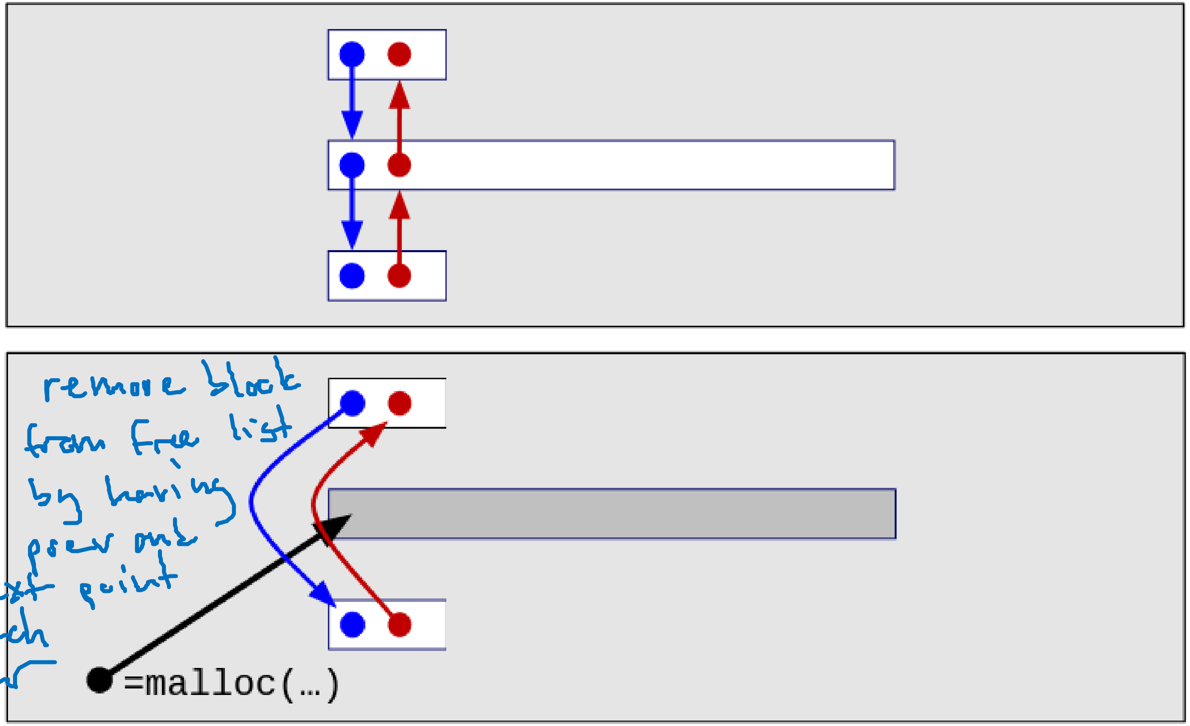
- **Physically**: blocks can be in any order
  - Free list ordering may not correlate to order in memory



Allocating From Explicit Free Lists

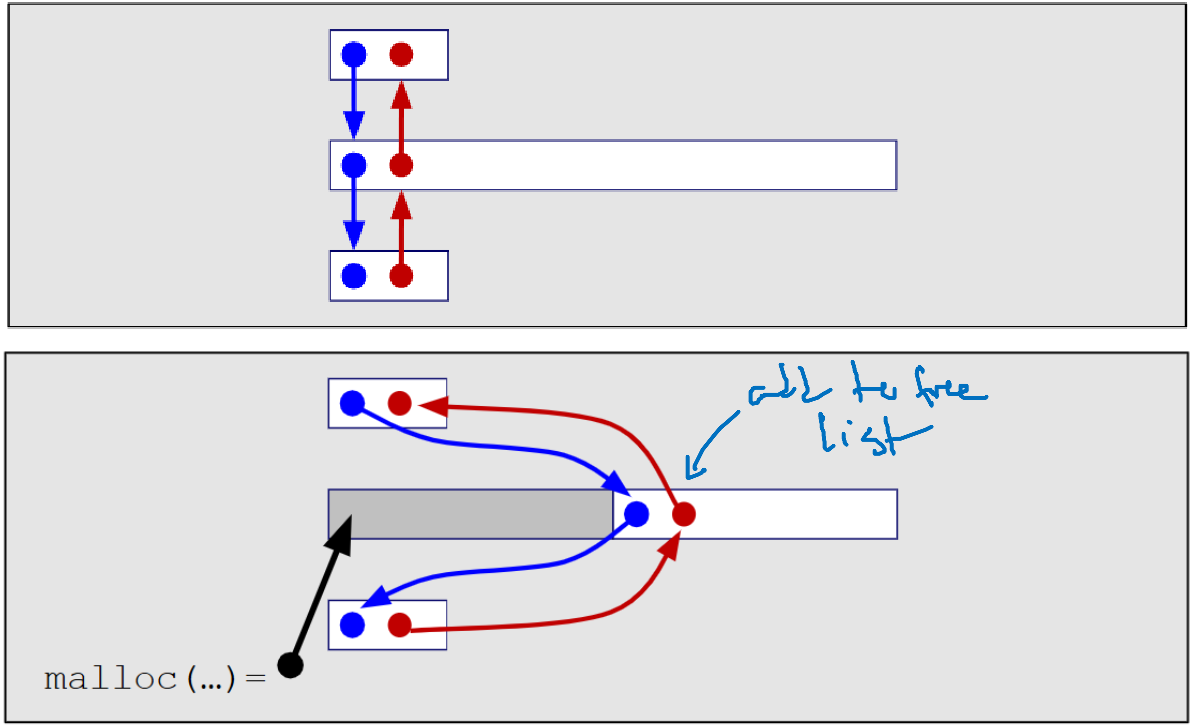
Note: diagram is not realistic

- Boundary tags omitted
- In reality, all pointers would point to the head of the block



remove block from free list by having prev and next point to each other

Allocating From Explicit Free Lists (pt 2)



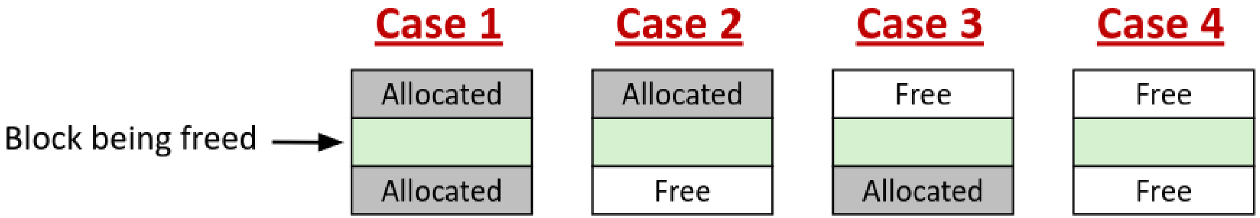
Freeing With Explicit Free Lists

- **Insertion policy**: when freeing block, where in the free list should it go?
  - **FIFO**: first-in, first-out
    - Insert new block at the head of the free list
    - Pros: simple. Insert blocks in constant time
    - Cons: research suggests worse fragmentation
  - **Address-ordered policy**
    - Insert block so that free list is in address order
    - Pros: research suggest less fragmentation
    - Cons: Insert blocks in linear time

Lab5 uses FIFO

Coalescing With Explicit Free Lists

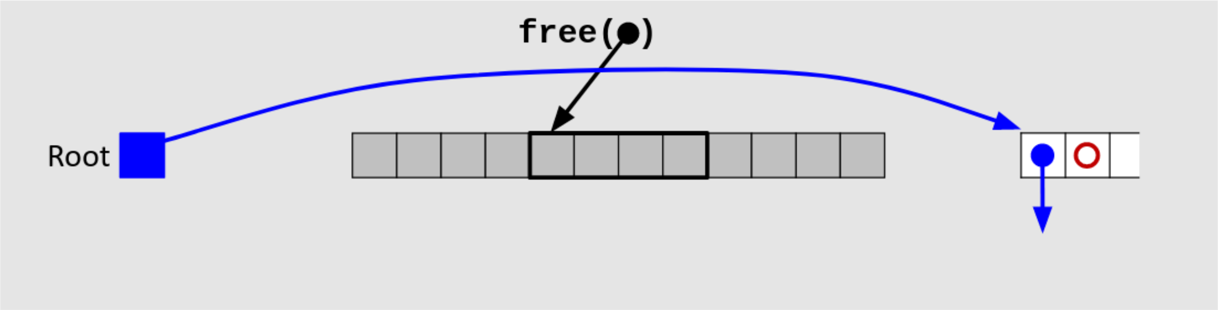
- Same cases as before

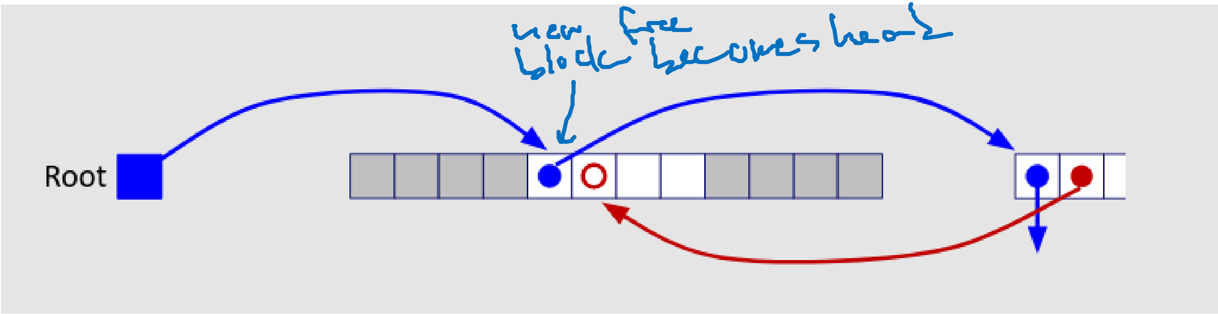


- Neighboring free blocks are already part of the free list
    1. Remove neighboring block(s) from free list
    2. Merge into a single, larger free block
    3. Add new block to the free list
- How do we know if a neighboring block is free?

Freeing Blocks with LIFO Policy (Case 1)

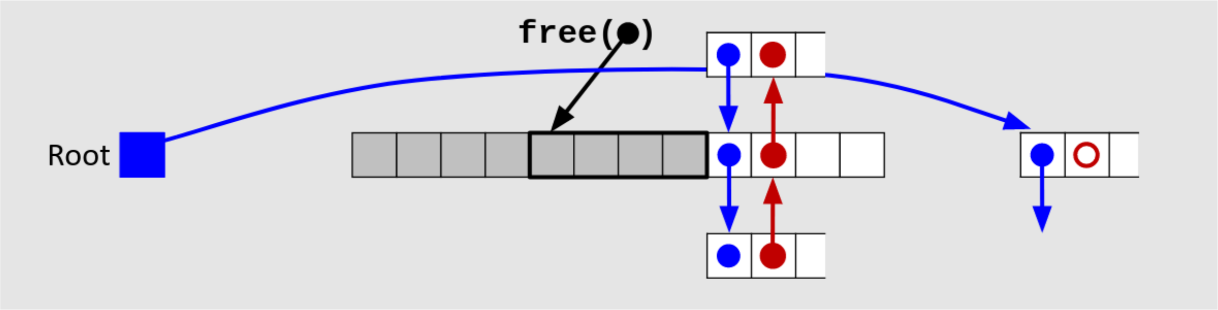
- No coalescing
- Newly freed block becomes list head
    - Old head becomes its next

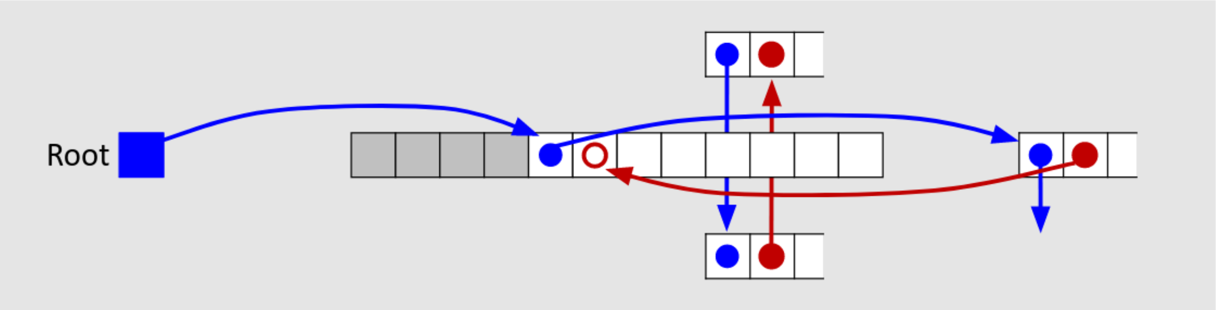




Freeing Blocks with LIFO Policy (Case 2)

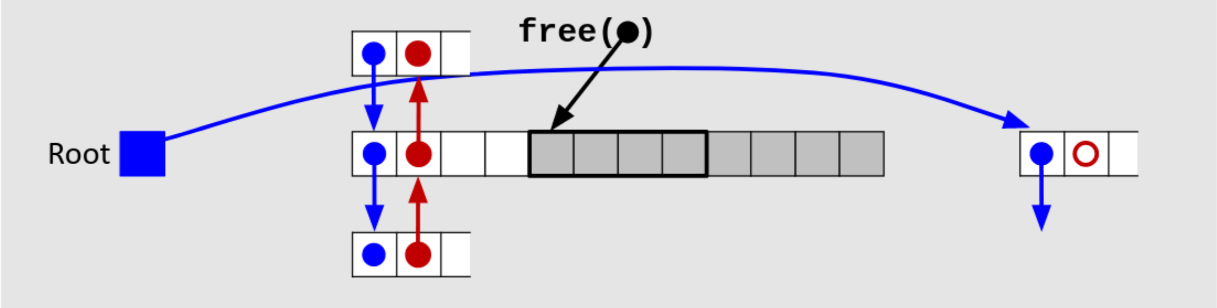
- Coalesce with following block
  - Following block gets removed from the list
- Newly made block becomes list head

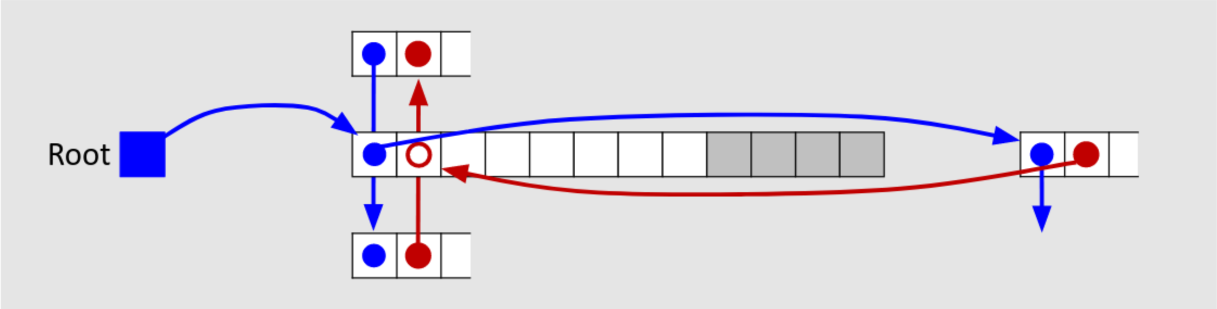




Freeing Blocks with LIFO Policy (Case 3)

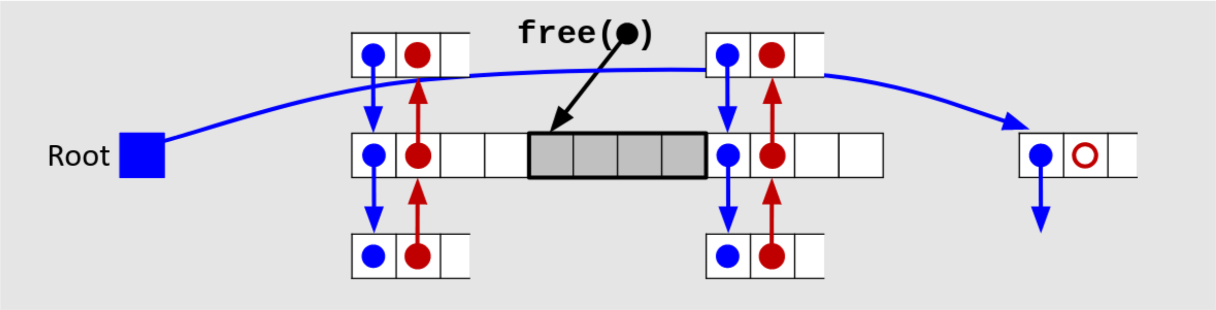
- Coalesce with preceding block
  - Preceding block gets removed from the list
- Newly made block becomes list head

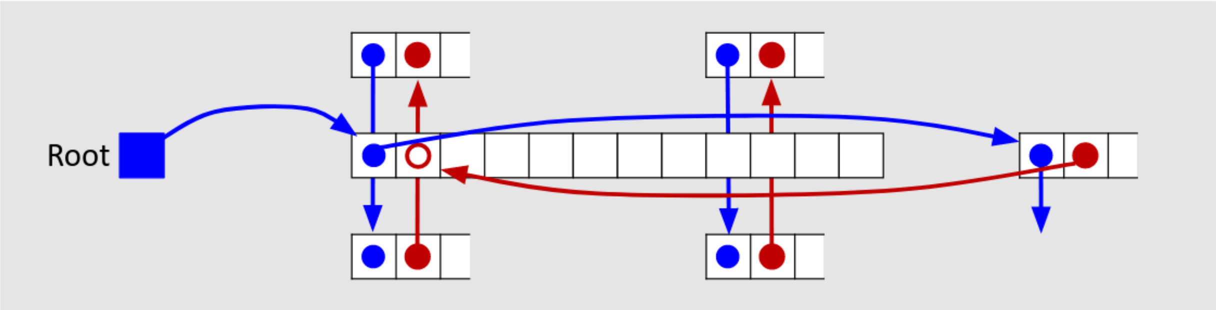




Freeing Blocks with LIFO Policy (Case 4)

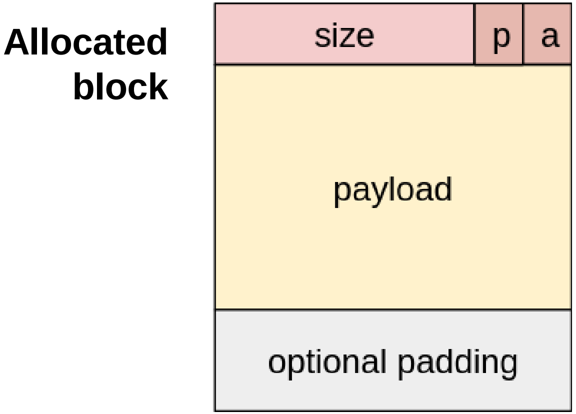
- Coalesce with both neighbors
    - Both neighbors get removed from the list
- Newly freed block becomes list head

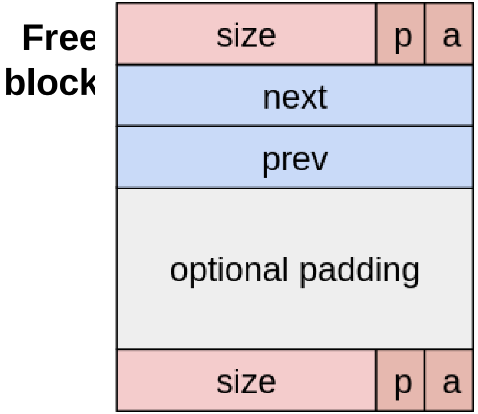




Do we always need boundary tags?

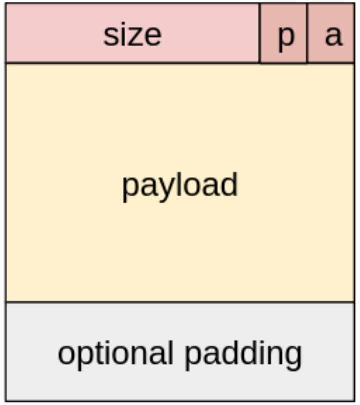
- Lab 5 suggests no… why not? (*Hint: when do we use boundary tags?)*
- We have room for more flags in our header!
  - Store another flag: **`preceding_allocated`**
  - If preceding block is allocated, don't coalesce with it

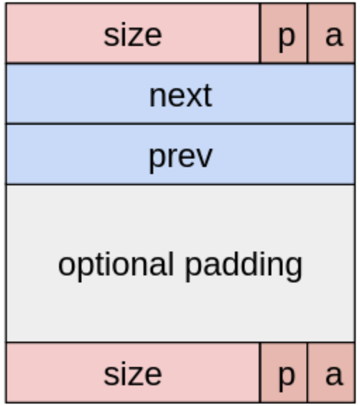




Explicit Free List Block Size

- Requirements for **allocated blocks**
  - Header (1 word)
  - Payload (1+ words)
  - **Minimum size = 2 words**
- Requirements for **free blocks**
  - Header (1 word)
  - Next pointer (1 word)
  - Prev pointer (1 word)
  - Footer (1 word)
  - **Minimum size = 4 words**

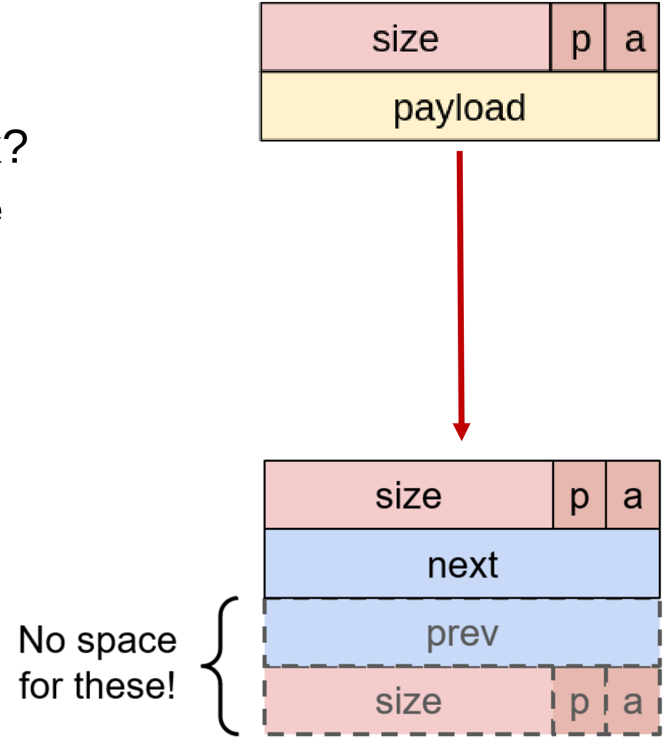




Explicit List Block Size (pt 2)

- **Problem**: what if we allocate a very small block?
  - When block is freed, we won't have room for the necessary fields

- **Solution**: never allocate a block smaller than the minimum free block size
  - Add padding to the end of allocated blocks
  - Don't split if resulting free block is too small



Dynamic Memory Allocation

- Overview
  - Introduction and goals
  - Allocation and deallocation (free)
- Malloc and free in C
  - Common memory-related bugs in C
- Fragmentation
- Explicit allocation implementation
  - Implicit free lists
  - Splitting and coalescing
  - Explicit free lists (Lab 5)
- **Implicit deallocation: garbage collection**

Wouldn’t it be nice…

- If we never had to free memory?
- Do you free objects in Java?
    - Reminder: *implicit* allocator

Garbage Collection

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

- Automatic reclamation of heap-allocated storage. *Application never frees memory!*
- Common in functional languages, scripting languages, and most modern object-oriented languages
  - Lisp, Racket, Erlang, ML, Haskell, Scala, Java, C#, Perl, Ruby, Python, Lua, JavaScript, Dart, Mathematica, MATLAB, many more…
- Variants (“conservative” garbage collectors) exist for C and C++
  - However, cannot necessarily collect all garbage

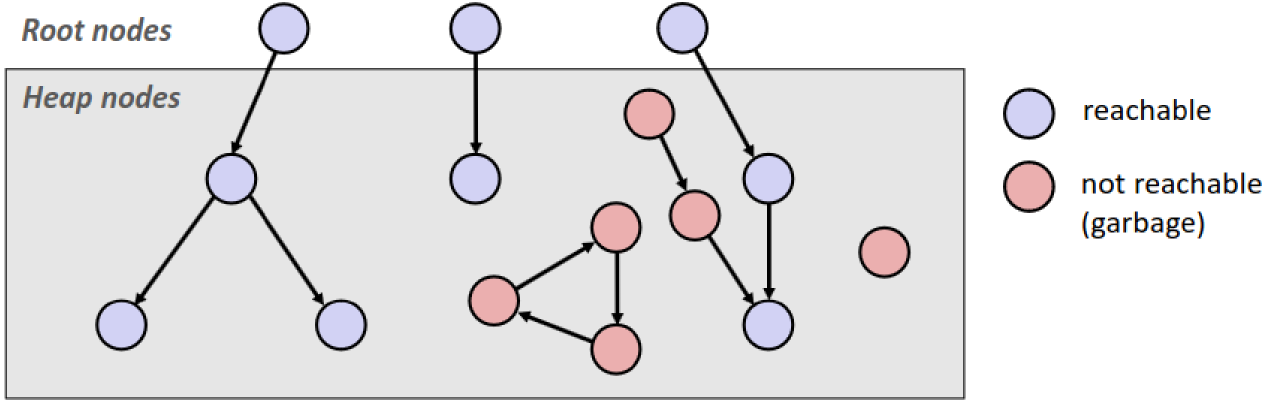
Garbage Collection (pt 2)

- How does the memory allocator know when memory can be freed?I
  - We cannot know what pieces of data are going to be used in the future
  - But, we can tell that certain blocks cannot be used if they are *unreachable* (via pointers in registers/stack/globals)
- If a program does not have any pointers to a block in the heap, then we know it can be cleaned up
  - Memory allocator needs to know what is a pointer and what is not – how can it do this?
    - Sometimes with help from the compiler

Memory as a Graph

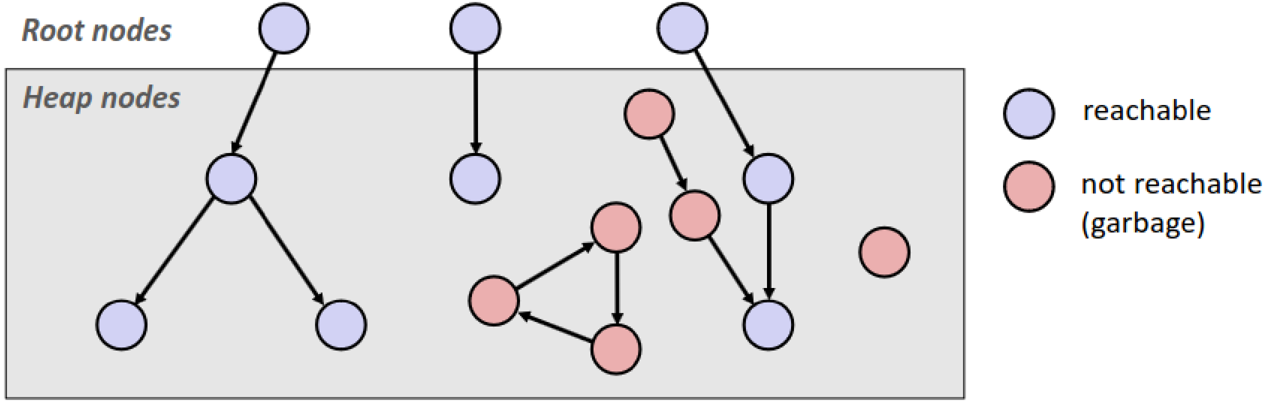
- We view memory as a directed graph
    - Each allocated heap block is a node in the graph
    - Each pointer is an edge in the graph
    - Locations not in the heap that contain pointers into the heap are called **root nodes** (e.g. registers, stack locations, global variables)

A node in the heap is **reachable** if there is a path from a root node to that node. Unreachable nodes are **garbage**.



Garbage Collection: Mark and Sweep

1. For every root node (i.e. every pointer in global data, the stack, or registers):
    a. If it is a pointer to a location on the heap, mark that heap block as **visited**
    b. Recursively go through pointers and mark them all as visited
2. Go through the entire heap:
    a. If a block is not marked, free it



Why doesn’t C have Garbage Collection

- People have tried! But it’s *impossible* to accurately predict what heap blocks C has access to. Why?
  - C allows you to “hide” pointers by casting them to another type
- Existing C garbage collectors are “conservative” (i.e. they may not free some blocks that could be freed)
  - Treat *every variable* as if it could be a pointer
  - Could cause memory leaks

Summary

- Three different policies for finding free blocks:
  - **First-fit**
  - **Next-fit**
  - **Best-fit**
- When free block is bigger than you need, **split** into two
- When freeing a block, **coalesce** with any adjacent free blocks in memory
  - Keep **`preceding_allocated`** bit in boundary tag
  - Store **footer** (copy of header) at the end of free blocks

Summary (pt 2)

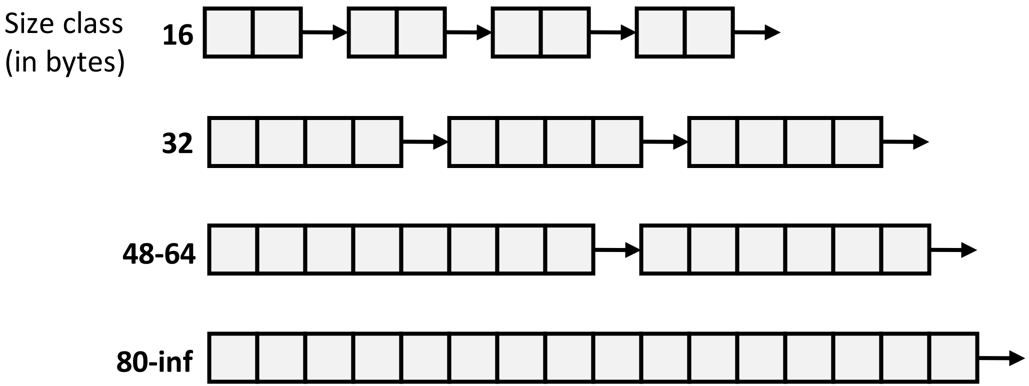
- Implicit free lists
  - Simpler to implement (no pointers to keep track of)
  - Less fragmentation
  - Slower, have to search through entire heap for free block
- Explicit free lists
  - Faster, only have to search through free blocks (instead of *all* blocks)
    - ***Much faster*** when most of the heap is full
  - More complicated to implement
  - Minimum block size is larger (free blocks need pointers) -> **more fragmentation**
  - In practice, often used in conjunction with *segregated free lists* (see bonus slides)
    - Keep a separate list for different block sizes/objects

BONUS SLIDES

The following slides are about the **SegList Allocator**, for those curious. You will NOT be expected to know this material.

Segregated List (SegList) Allocators

- Each *size class* of blocks has its own free list
- Organized as an array of free lists
- Often have separate classes for each small size
- For larger sizes: One class for each two-power size



SegList Allocator

- Have an array of free lists for various size classes

- To free a block:
  - Mark block as free
  - Coalesce (if needed)
  - Place on appropriate class list

SegList Advantages

- Higher throughput
  - Search is log time for power-of-two size classes

- Better memory utilization
  - First-fit search of seglist approximates a best-fit search of entire heap
  - *Extreme case:* Giving every block its own size class is no worse than best-fit search of an explicit list
  - Don’t need to use space for block size for the fixed-size classes