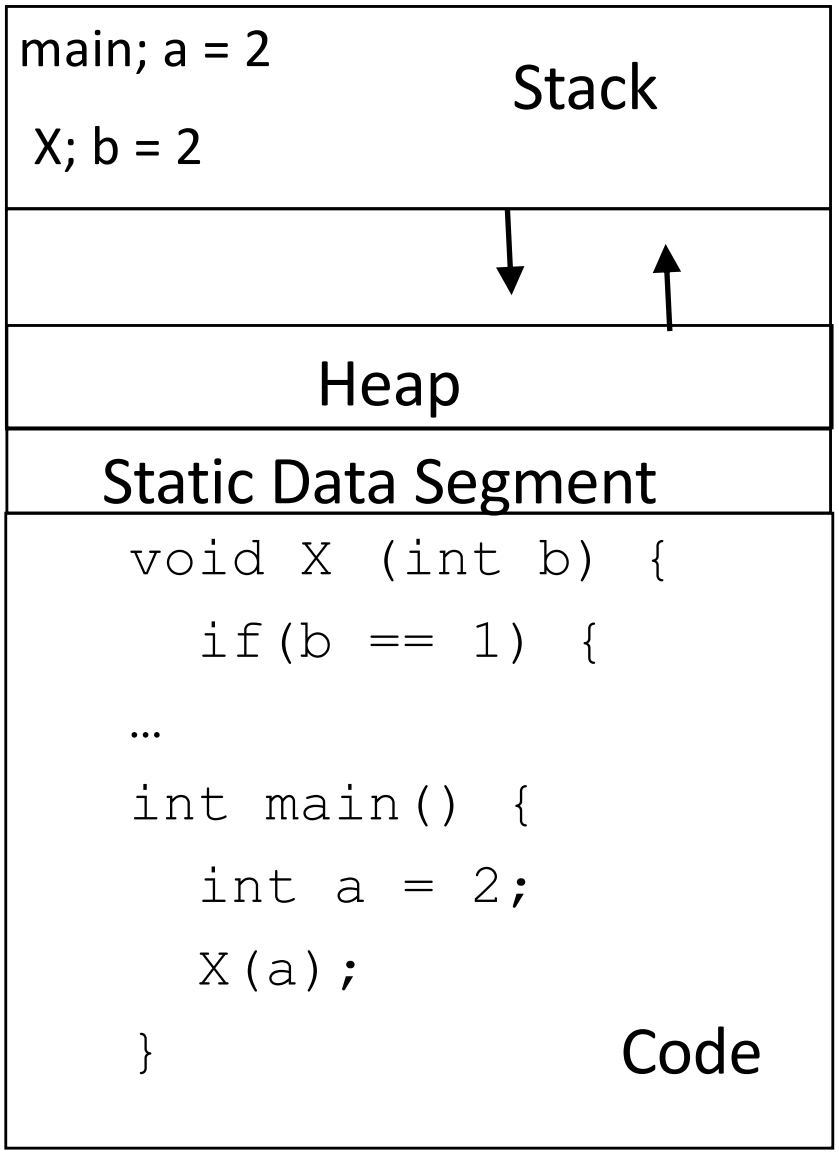
Heap Memory Management

Bringing It Together

- Processes are the OS's main abstraction for protection
- A process defines an address space---but this is a virtual/logical address space
  - Virtual address spaces are mapped onto physical address spaces in physical memory
  - The VAS is divided up into equally-sized units called pages
- The VAS has segments: code, SDS, stack, and *heap*

Reminder


main; a = 2
X; b = 2
Stack
Heap
Static Data Segment
void X (int b) {
if(b == 1) {
…
int main() {
int a = 2;
X(a);
}
Code

What’s in the heap?

Heap Memory Manager

Heap Memory Management

- The runtime system manages the heap
  - Program/runtime system requests space for the heap from the OS
    - At the beginning *and* when the heap is full and more space is needed
  - OS gives memory to a process 1 to *k* pages at a time
  - Typically, the runtime system will not “free” the pages
  - Reclaimed when by the OS when the program ends
- How does the runtime system *efficiently* allocate and recycle memory on behalf of the program?

Goals for the Heap Memory Manager

- Allocate the space
  - Be fast!
  - Keep memory space organized
  - Low fragmentation (wasted space)
- <User uses it>
- Deallocate the space
  - Be fast!
  - Reclaim the space for later use

Runtime System Requirements

- Handle arbitrary request sequences
  - Memory may be allocated and freed in any order
- Make immediate responses to requests
  - Cannot reorder/buffer requests to improve performance
- Use only the heap
  - Any data structures (such as free list) used by `malloc()/free()` must be stored on the heap
- Align blocks (e.g., on 8-byte boundary)
  - Blocks must be able to hold any type of data object
- Not modify allocated blocks
  - Can only manipulate or change free blocks
  - Cannot modify (or move!) other blocks after they are allocated
  - Results in fairly simple allocation policies

Explicit vs. Automatic Memory Management

Two Categories of Heap Memory Management

Explicit memory management

- The program(mer) explicitly manages all of the memory
  - Allocation: malloc/new
  - Deallocation: free/delete
- Pointers: anything may or may not be a pointer
- Example languages: C, C++

Explicit Memory Management: Division of Labor

- User
  - Explicitly allocates memory by requesting a number of bytes
  - May explicitly request deallocation of memory when it is no longer used
- Runtime System
  - Receives requests for memory
  - Identifies appropriate location for allocation
  - Returns pointer
  - Later, frees allocation on request

Explicit Memory Management Characteristics

- Programmer burden can be high
  - More code to maintain
  - Correctness can be elusive
    - Free an object too soon -> core dump
    - Free an object too late -> waste space
    - Never free -> at best waste, at worst fail
- Efficiency can be very high
- Gives programmers control

Two Categories of Heap Memory Management

Automatic memory management

- The program(mer) explicitly allocates memory, but the runtime system manages it
  - Allocation: new
  - Deallocation: *None*
- Pointers: Program and runtime system know all pointers
- Runtime system must discriminate live (in use) objects and garbage
- Example languages: Java, Python

Automatic Memory Management Characteristics

- Reduces programmer burden
  - Less user code compared to explicit memory management
  - Protects against some classes of memory errors
    - No free(), thus no premature free(), no double free(), or forgetting to free()
  - Not perfect, memory can still leak
- Efficiency can be lacking
- Reduced programmer control

Garbage Collection Brought to You by…

- “Safe” pointers!
  - Used in managed languages that provide garbage collection
  - Programs may not access arbitrary addresses in memory
  - The compiler can identify and provide to the garbage collector all the pointers, thus
  - “Once garbage, always garbage”
  - Runtime system can move objects by updating pointers

Allocation Techniques: Bump Pointer

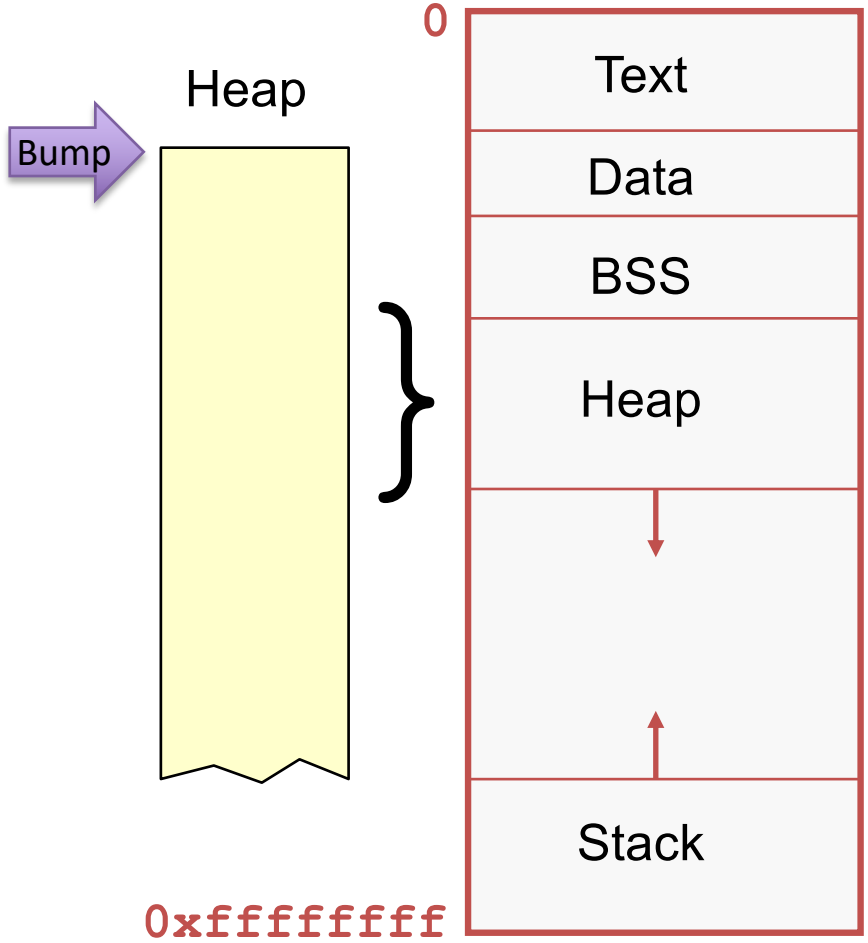
Allocation Techniques

1. Bump-pointer
   - Contiguous allocation (for all requested blocks)
   - Pointer begins at start of heap
   - As requested, bytes allocated, and pointer is "bumped" past allocation

Bump Pointer

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

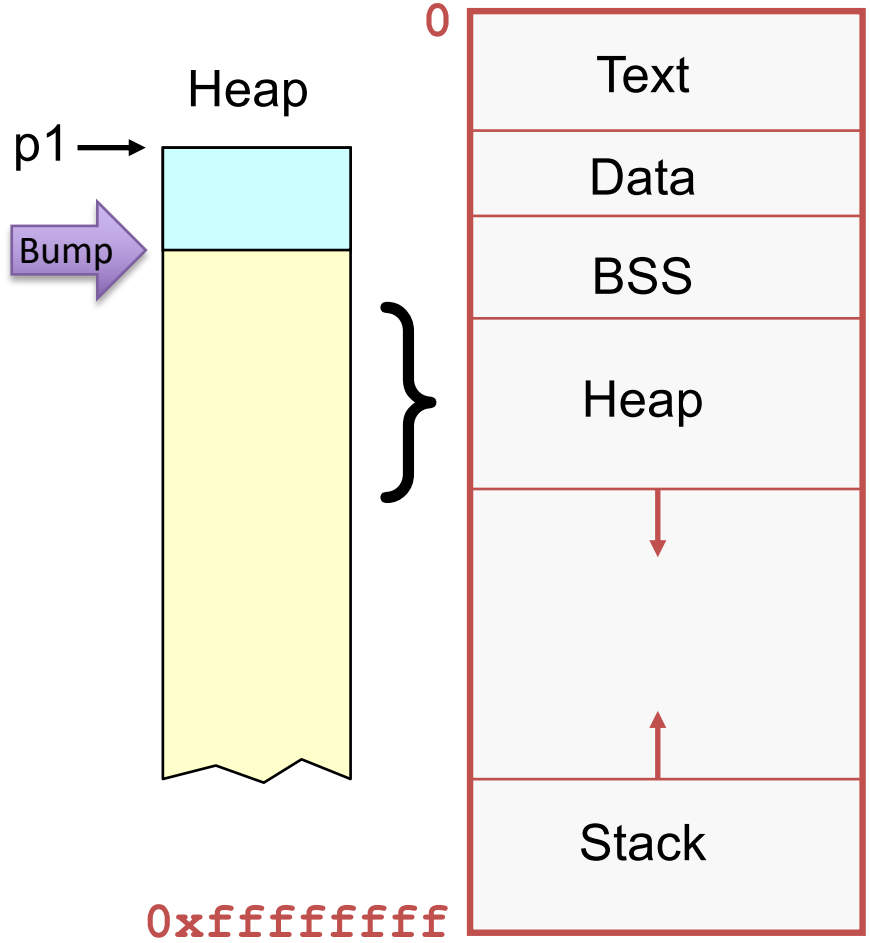
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Bump Pointer

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

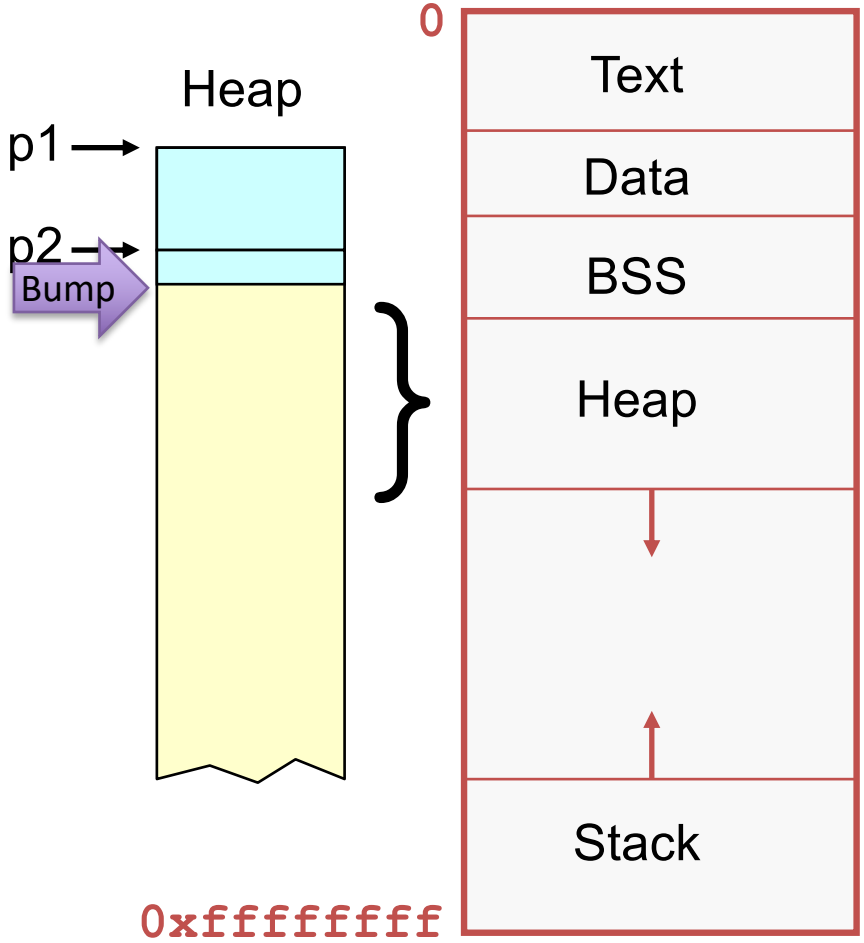
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Bump Pointer

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

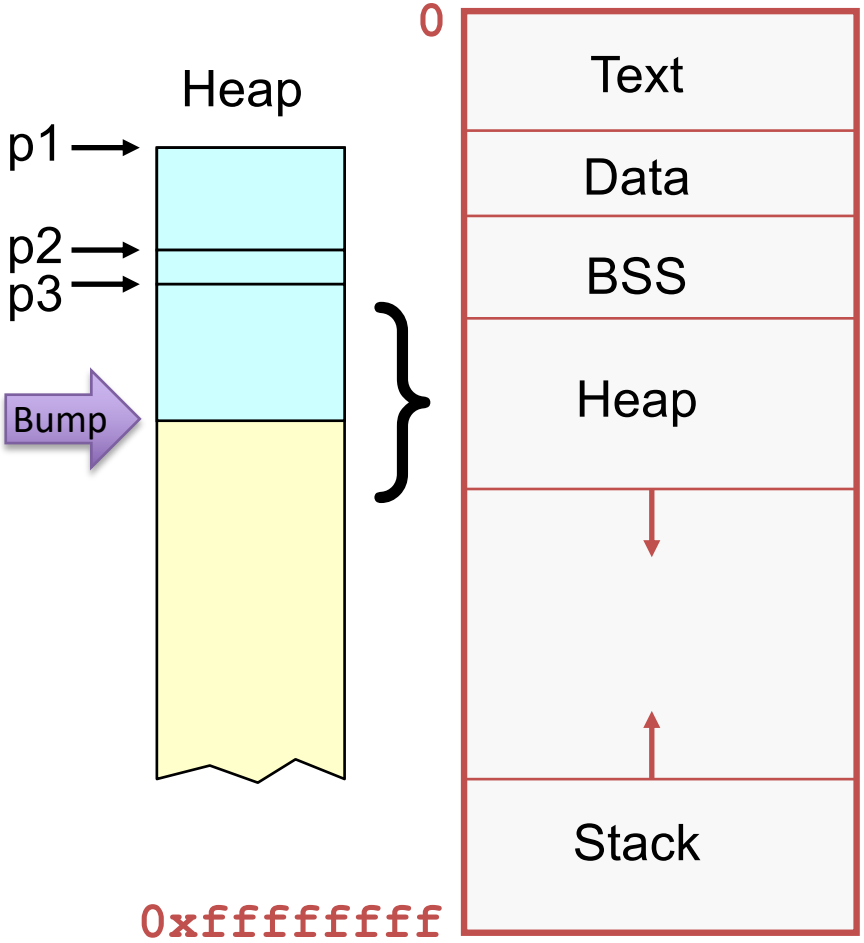
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Bump Pointer

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

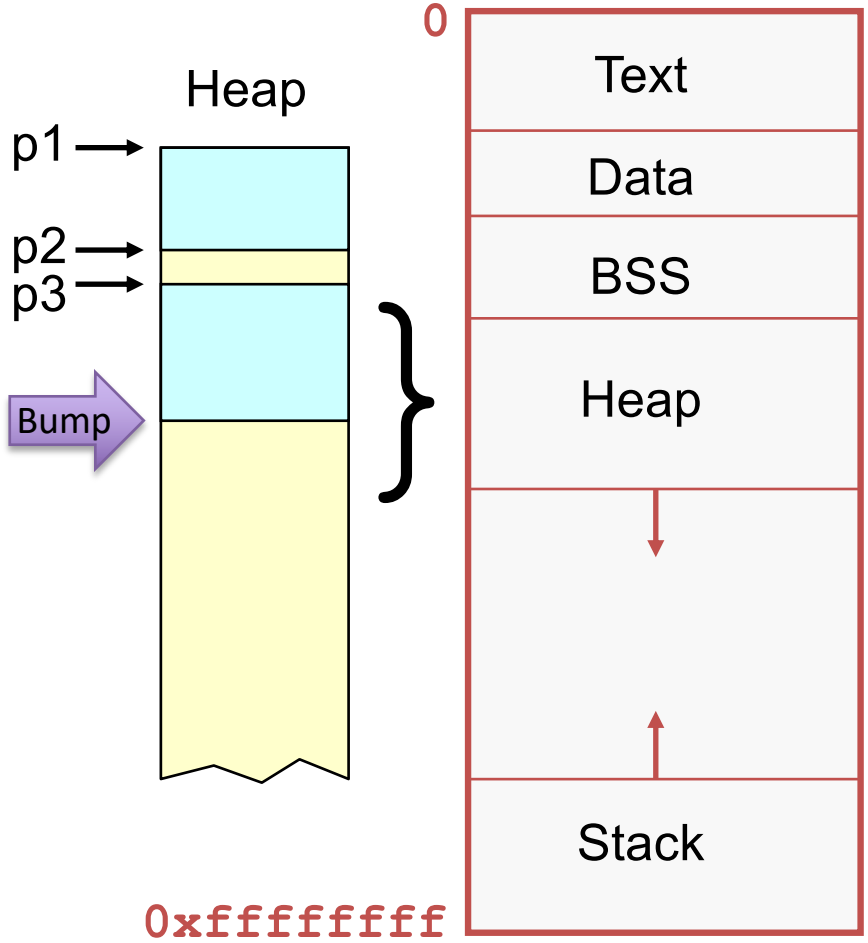
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Bump Pointer

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

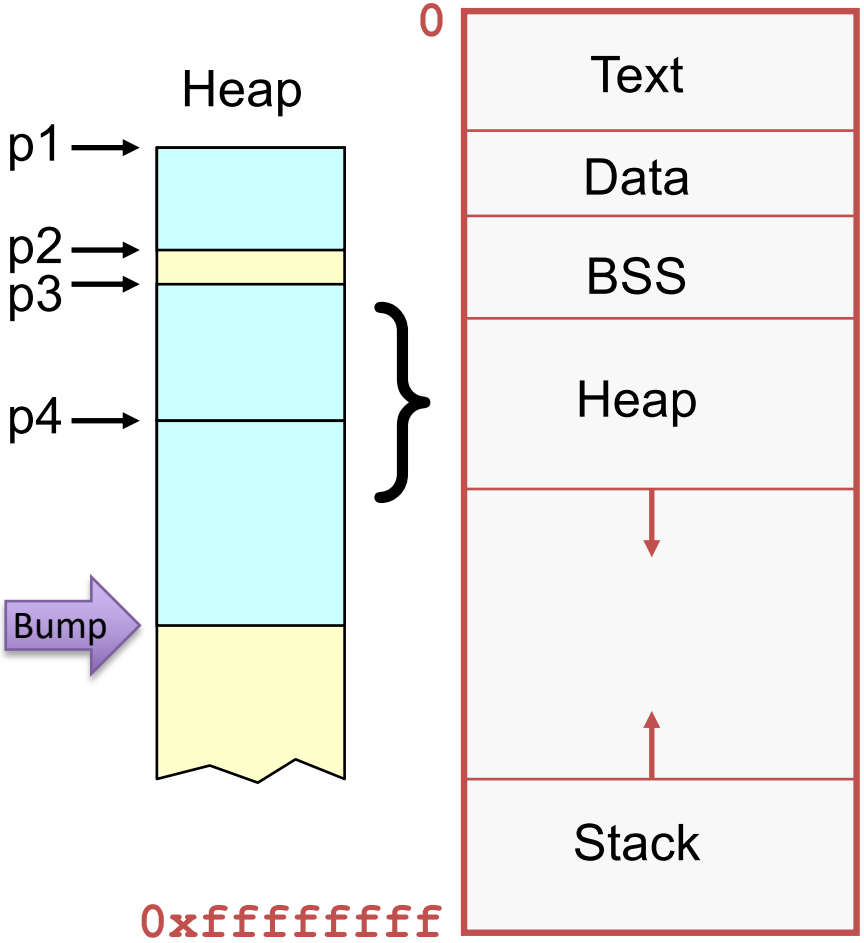
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Bump Pointer

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

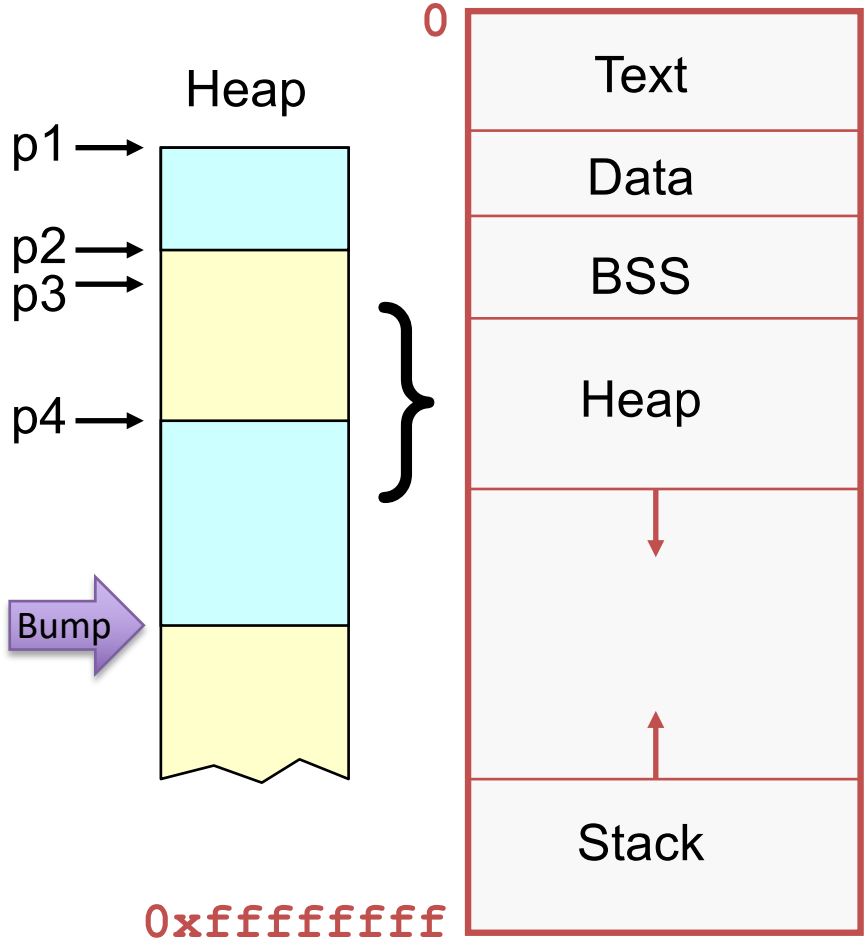
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Bump Pointer

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

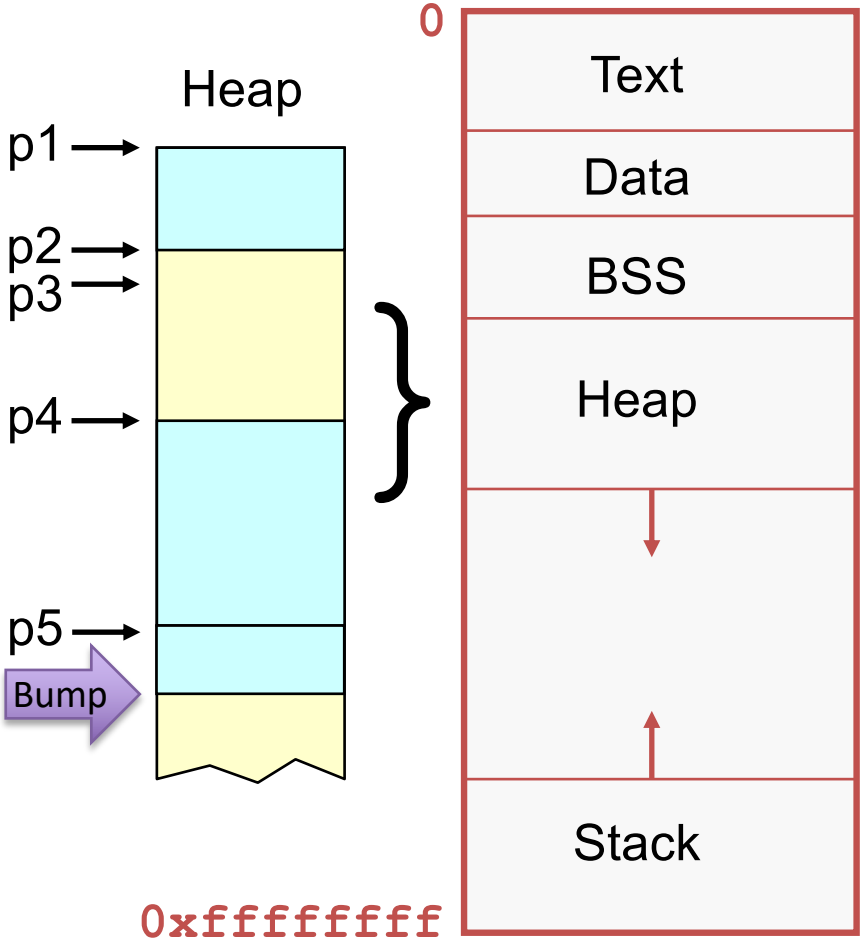
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Bump Pointer

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

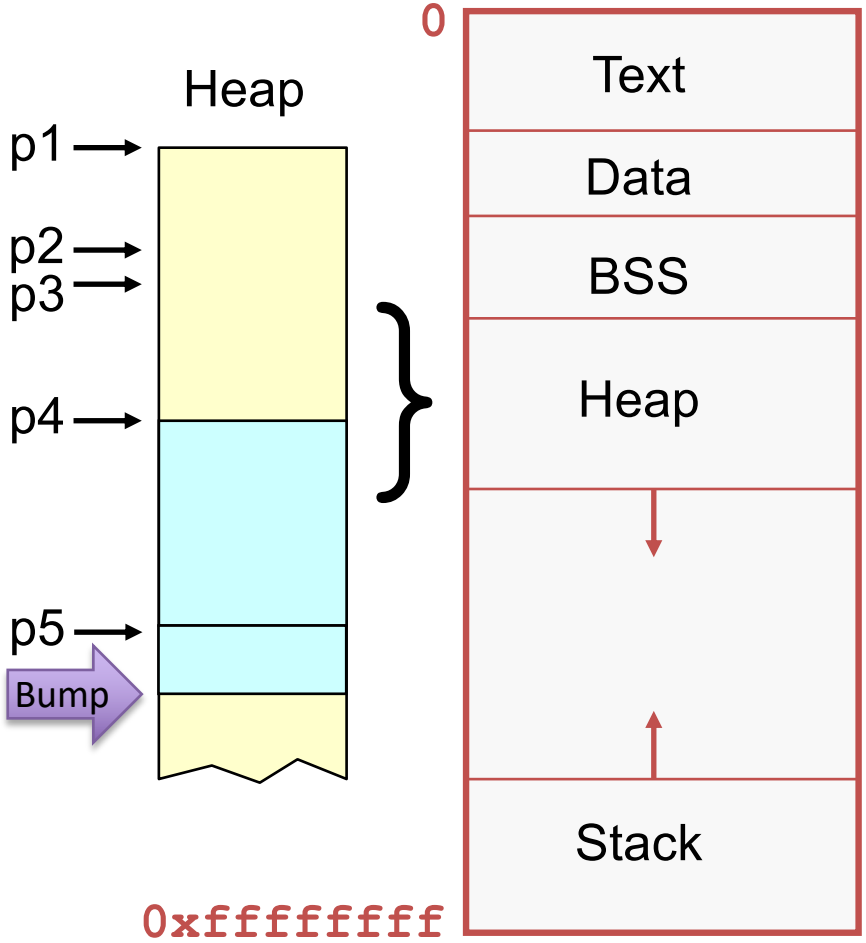
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Bump Pointer

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

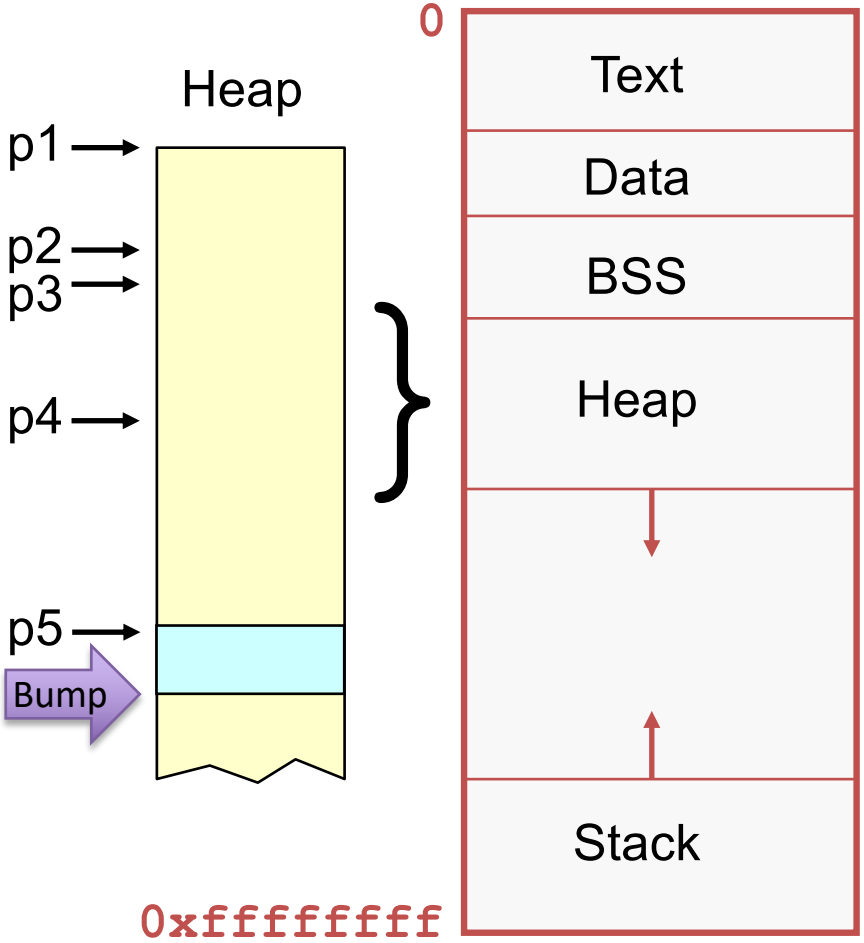
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Bump Pointer

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

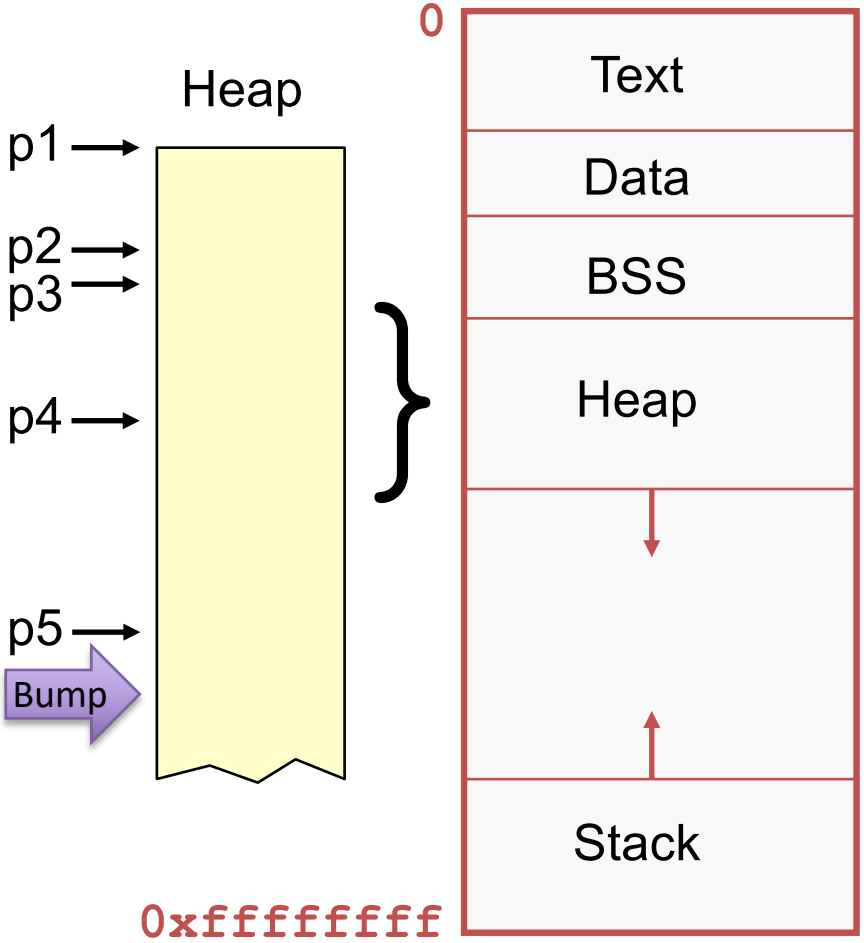
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Bump Pointer

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Bump Pointer: Text Description

- In this address space, the heap grows downwards towards higher addresses
  - order of segments starting at address 0: text, Data, BSS and heap
    - stack starts at some high address later and grows toward lower addresses
    - note: stack and heap grow towards each other
- List of C commands and their consequences:
  - note: originally heap is empty and bump pointer at address 100
  - char *p1 = malloc(3)
    - 3 bytes are allocated on the heap
    - p1 holds address 100, the beginning of the 3 bytes
    - bump pointer moved to address 103
  - char* p2 = malloc(1)
    - 1 byte is allocated on the heap
    - p2 holds address 103, the beginning of the 1 byte
    - bump pointer moved to address 104
  - char* p3 = malloc(4)
    - 4 bytes are allocated on the heap
    - p3 holds address 104, the beginning of the 4 bytes
    - bump pointer moved to address 108
  - free(p2)
    - memory that p2 points to is deallocated
    - now is a gap in the heap from 103 to 104
      - gap stays because bump pointer allocation doesn't move memory around once it has been allocated
  - char* p4 = malloc(6)
    - 6 bytes allocated on the heap
    - p4 holds address 108, the beginning of the 6 bytes
    - bump pointer moved to address 114
  - free(p3)
    - memory that p3 points to is deallocated
    - now is a gap in the heap from 103 to 108
      - this new gap is coalesced with the old gap created by freeing p2
  - char* p5 = malloc(2)
    - 2 bytes allocated on the heap
    - p5 points to address 114, the beginning of the 2 bytes
    - bump pointer is moved to address 116
  - free(p1)
    - memory that p1 points to is deallocated
    - now a gap in heap from 100 to 108
  - free(p4)
    - memory that p4 points to is deallocated
    - now gap in heap from 100 to 114
  - free(p5)
    - memory that p5 points to is deallocated
    - now a gap in heap from 100 to 116
  - now: all of the memory that was originally allocated on the heap has been reclaimed
  - note: pointers still hold their old addresses because they were not NULLed out
    - this means can still try to use them but will get a segfault

Allocation Techniques: Free List

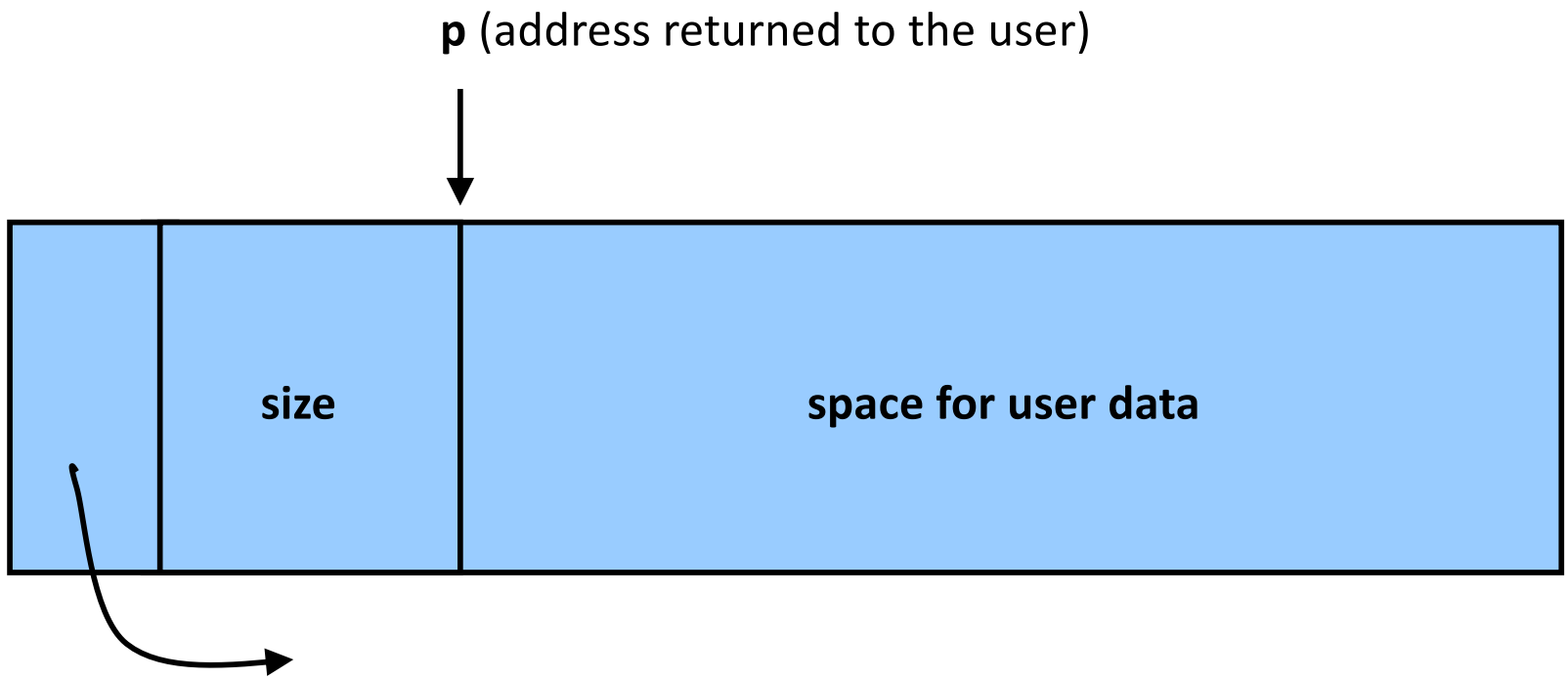
Allocation Techniques

2. Free List
   - Divides memory into some size blocks
   - Maintains a free list
     - Must be stored in the heap
     - Uses a special structure for free blocks
   - To allocate memory, find block in the free list
     - Using what algorithm? Guess!
     - If the right size does not exist, carves up a bigger piece
   - To deallocate memory, put memory back on the free list

Free Block: Pointer, Size, Data

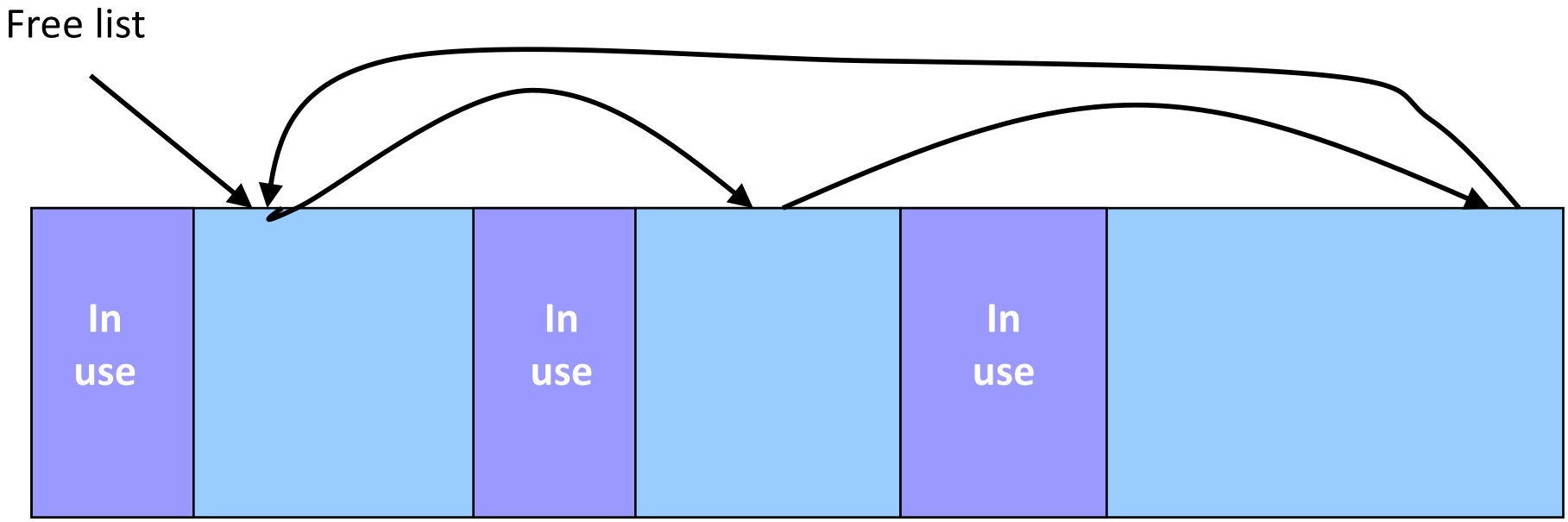
Free block in memory

- Pointer to the next free block
- Size of the free block
- Free space (that can be allocated to user)



Free List: Circular Linked List

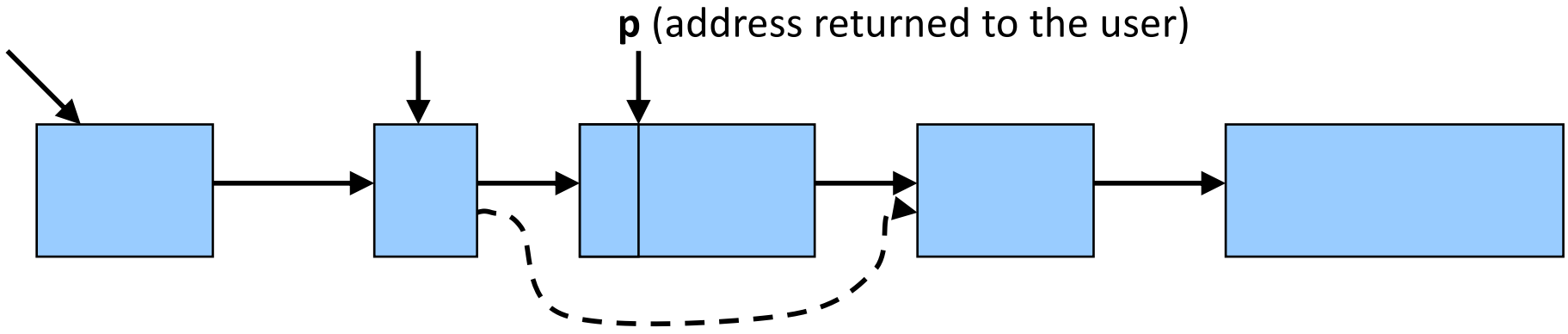
- Free blocks, linked together
- List may be ordered by address or by size, depending on allocation algorithm



Choosing the Spot
First Case: A Perfect Fit

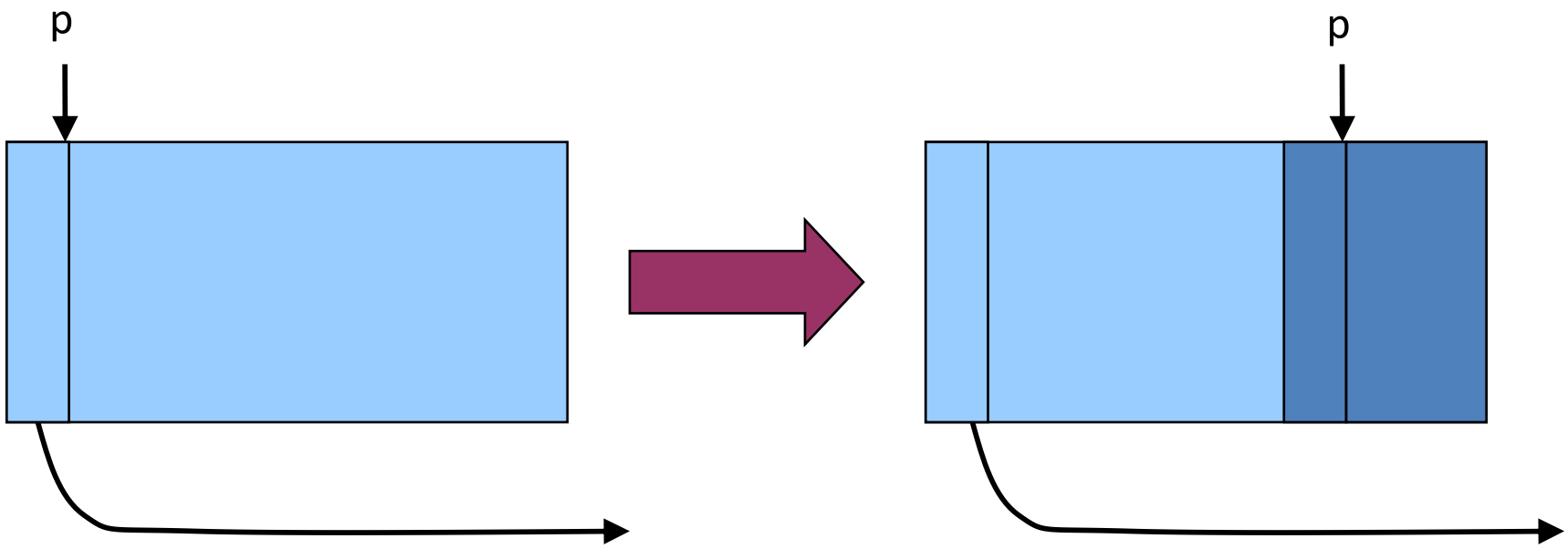
Suppose the block is a perfect fit

- Remove the element from the list
- Link the previous element with the next element
- Return the current element to the user
  - Skipping header with metadata



Choosing the Spot
Second Case: Block is Too Big

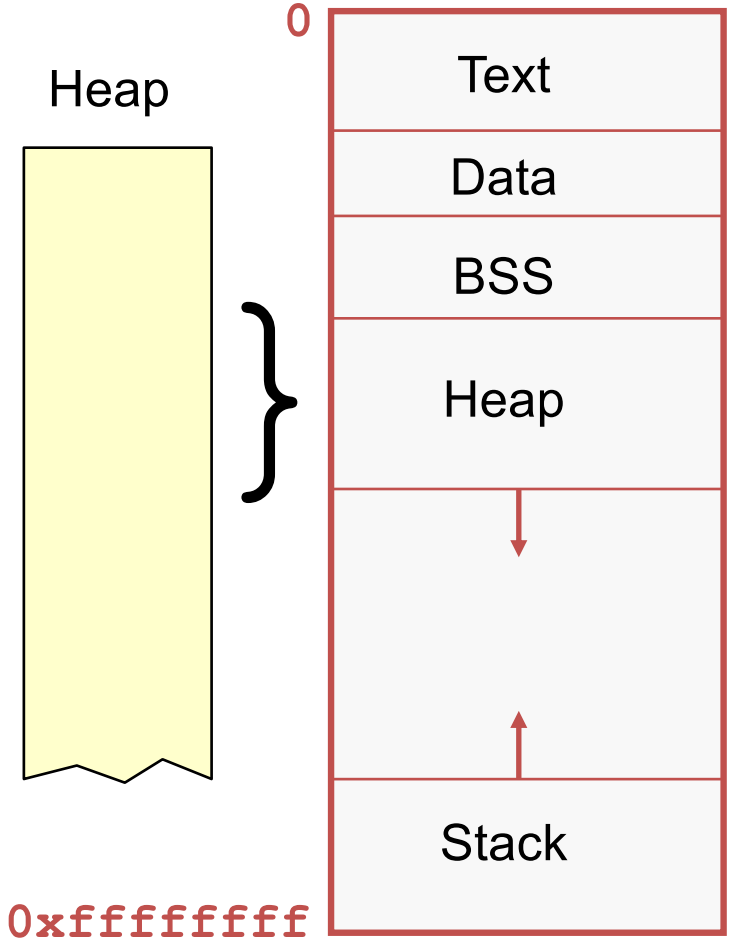
- Suppose the block is bigger than requested
  - Divide the free block into two blocks
  - Keep first (now smaller) block in the free list
  - Allocate the second block to the user



Free List

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

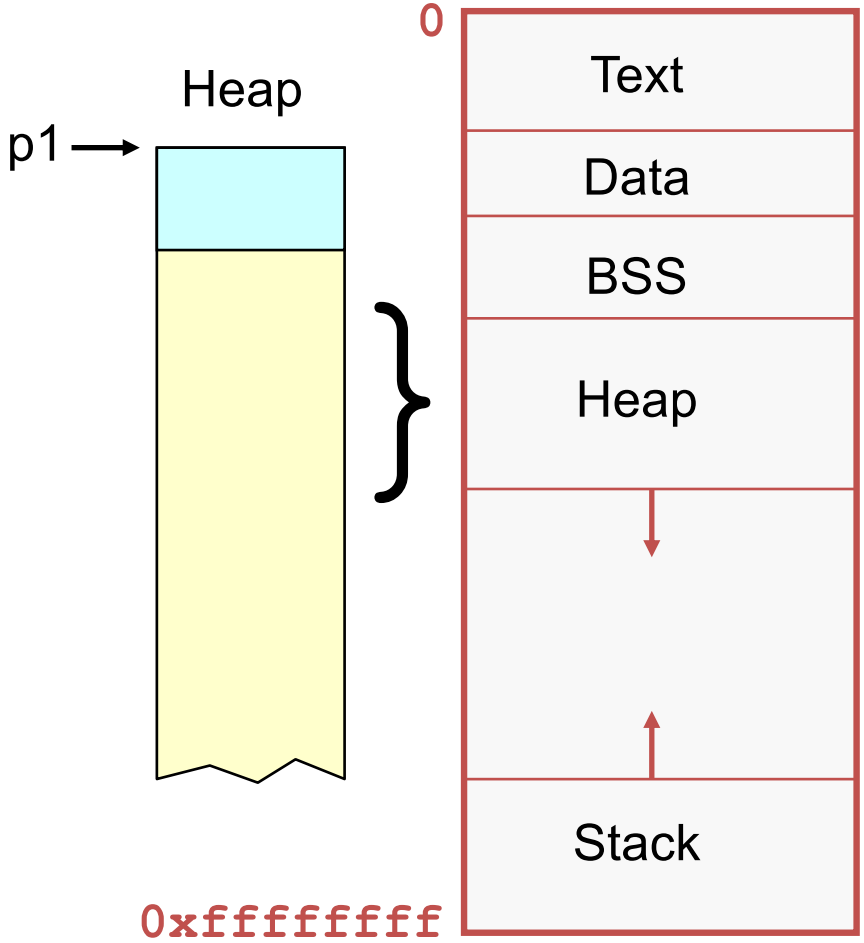
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Free List

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

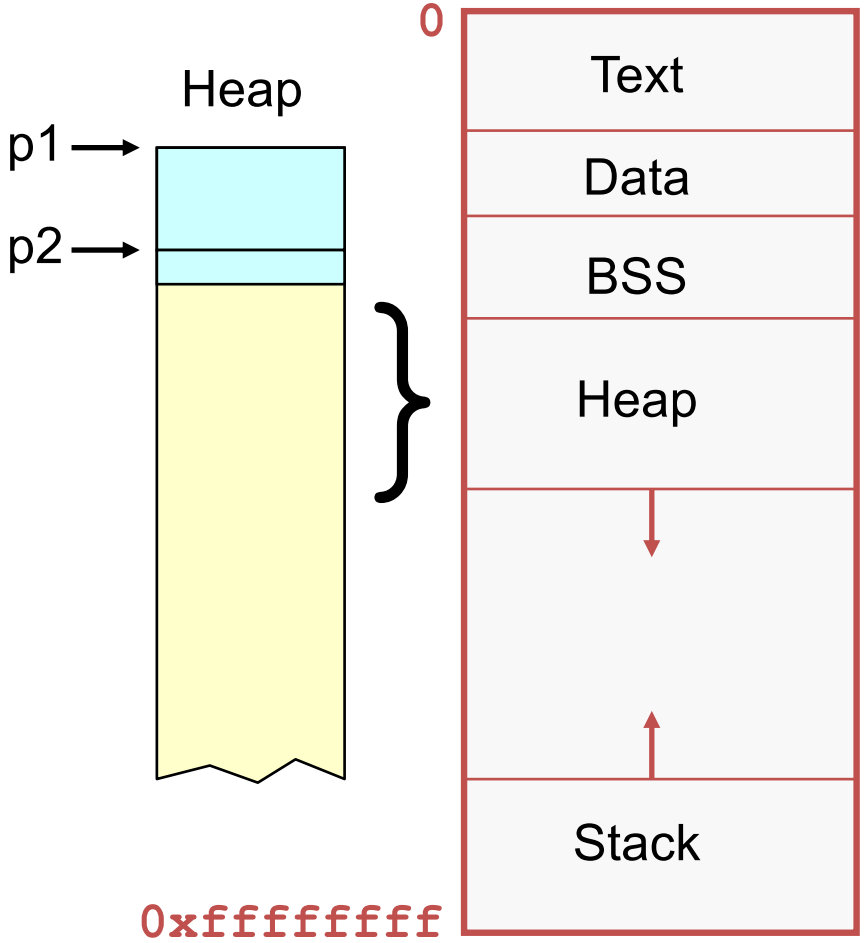
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Free List

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

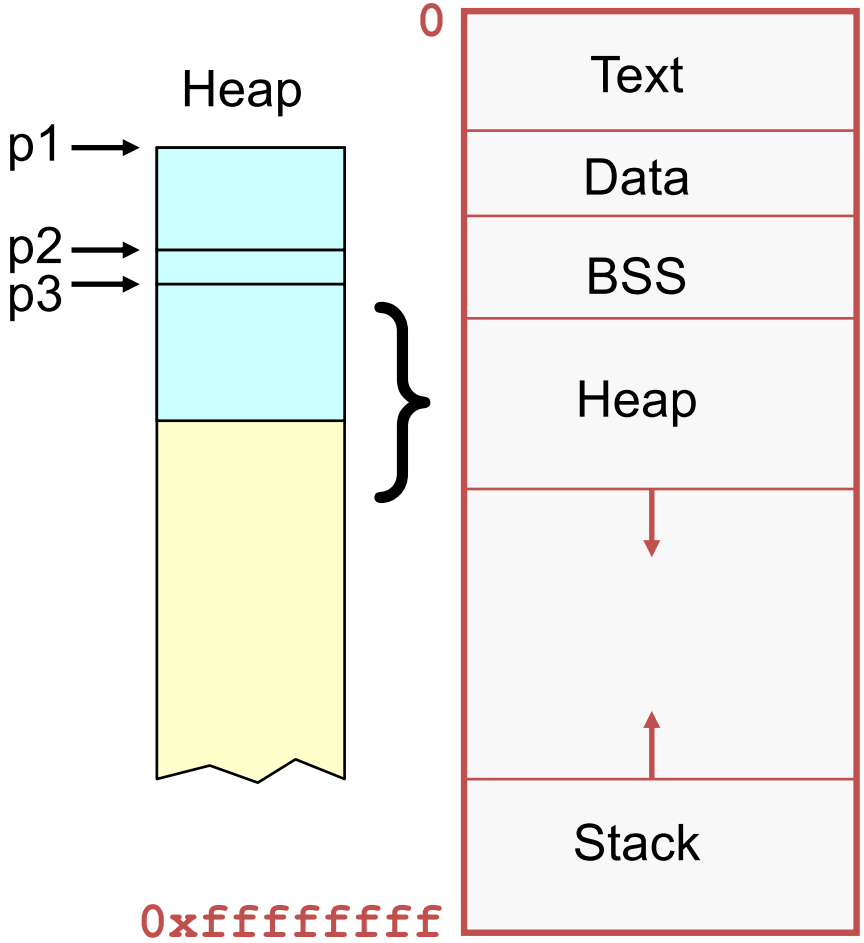
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Free List

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

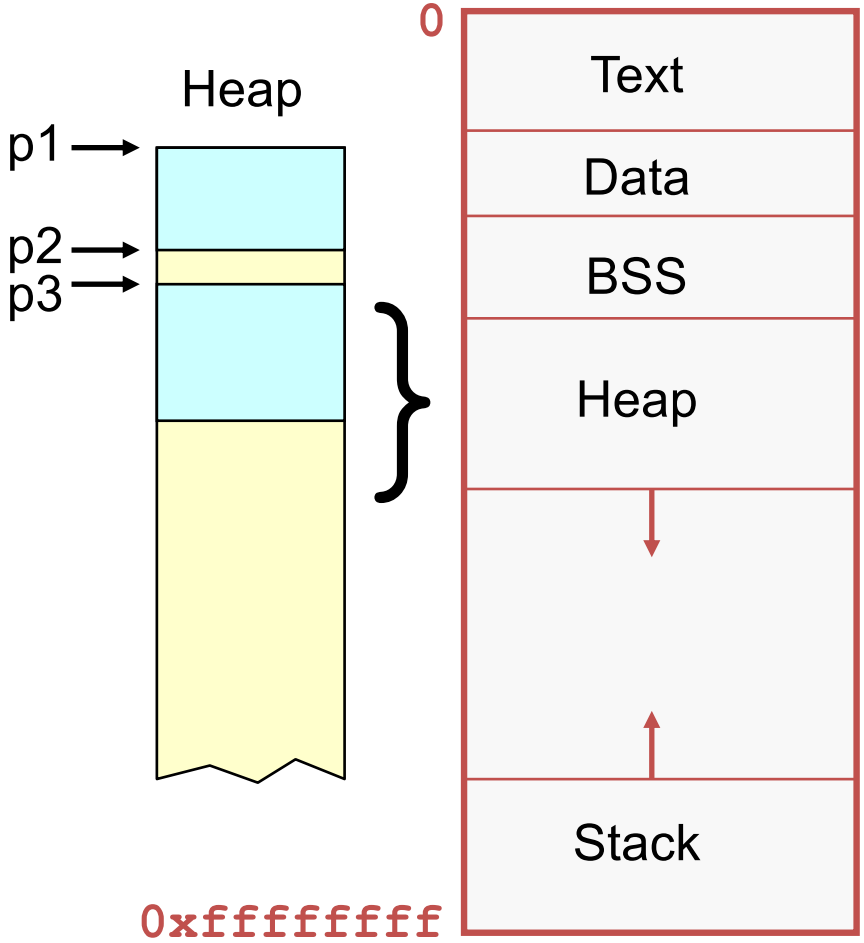
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Free List

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

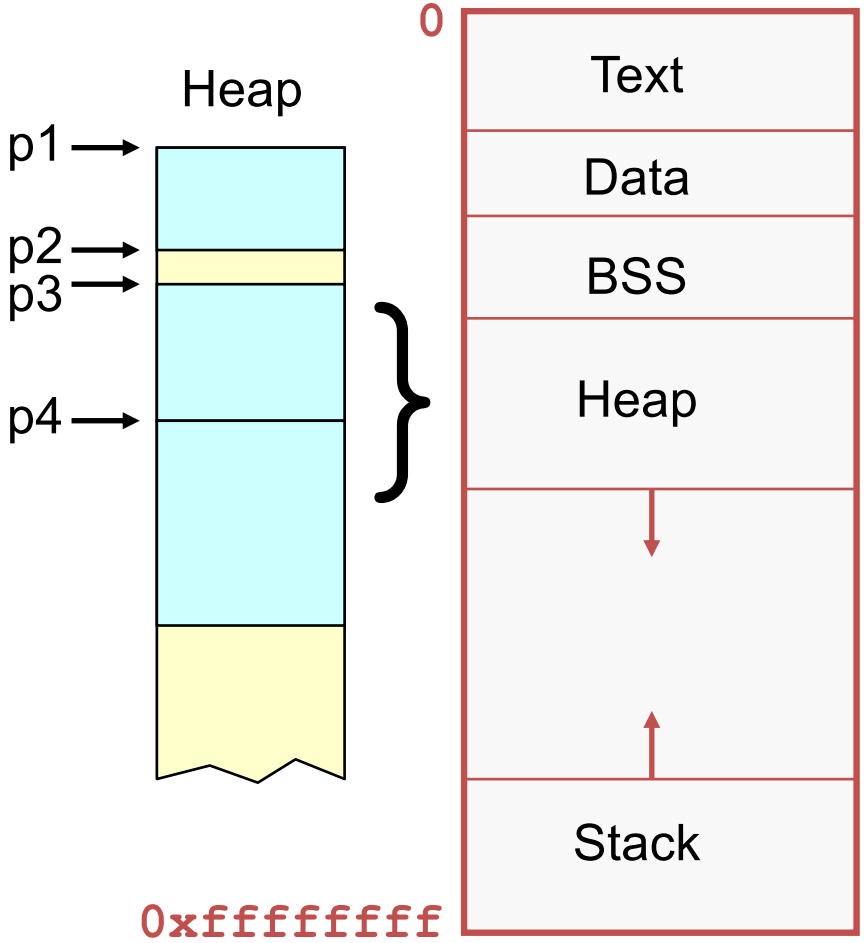
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Free List

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

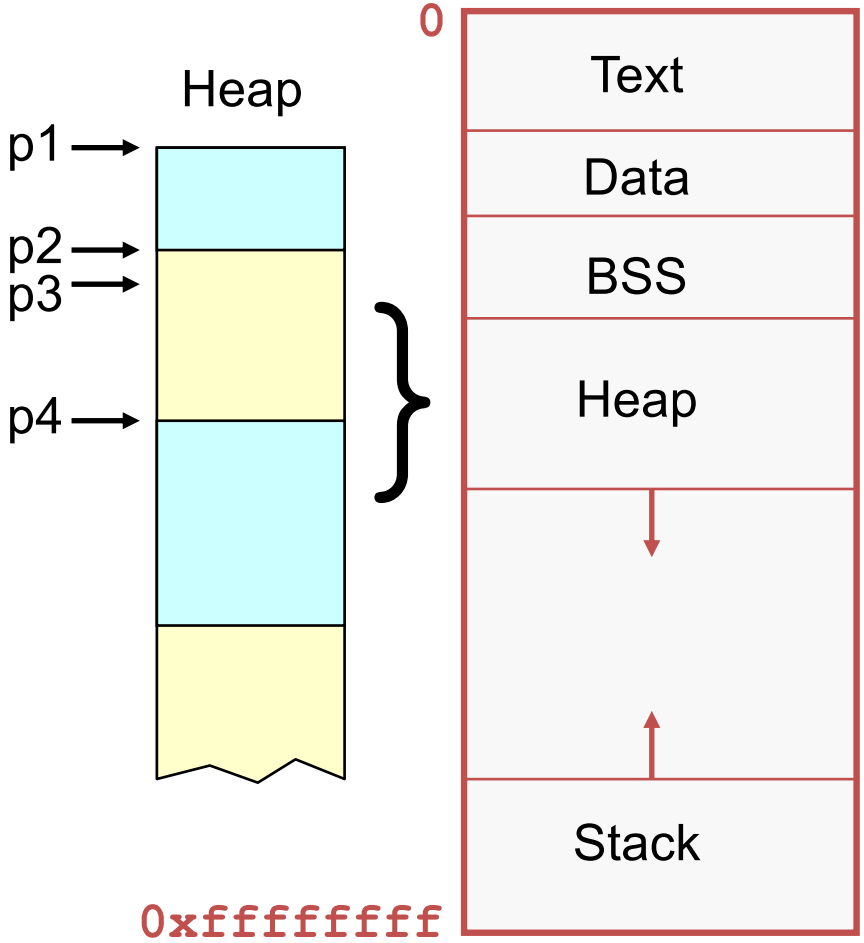
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Free List

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

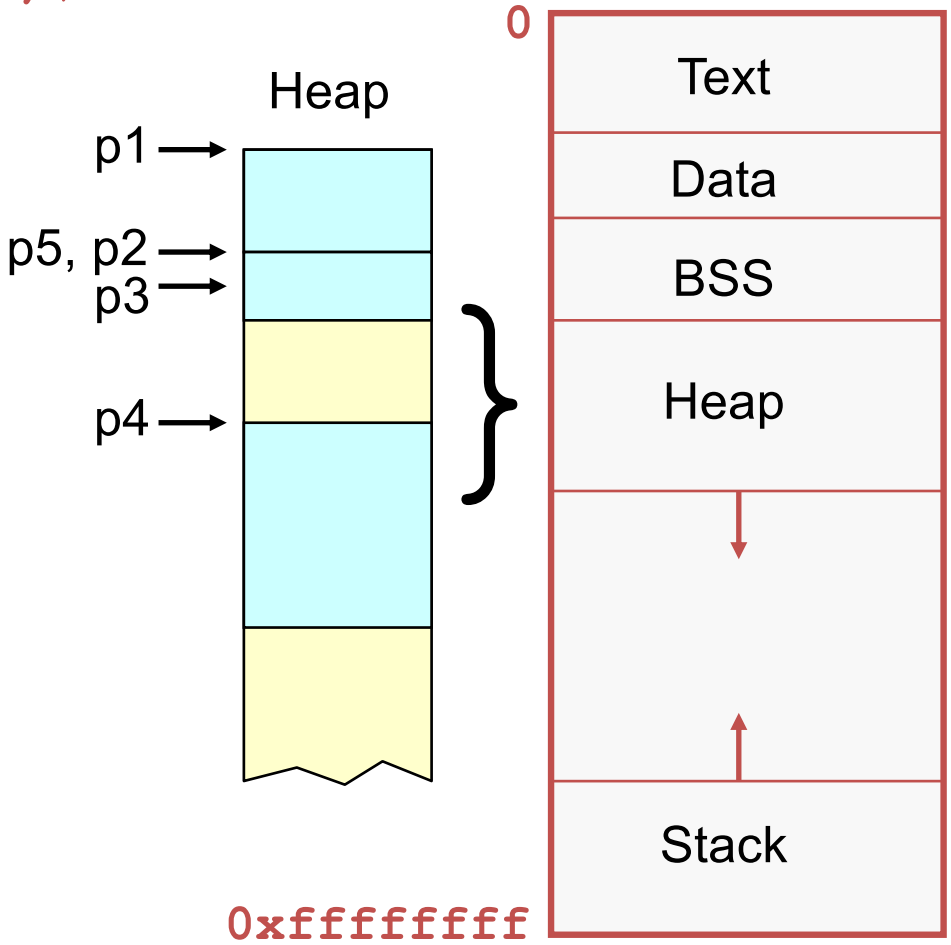
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Free List

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

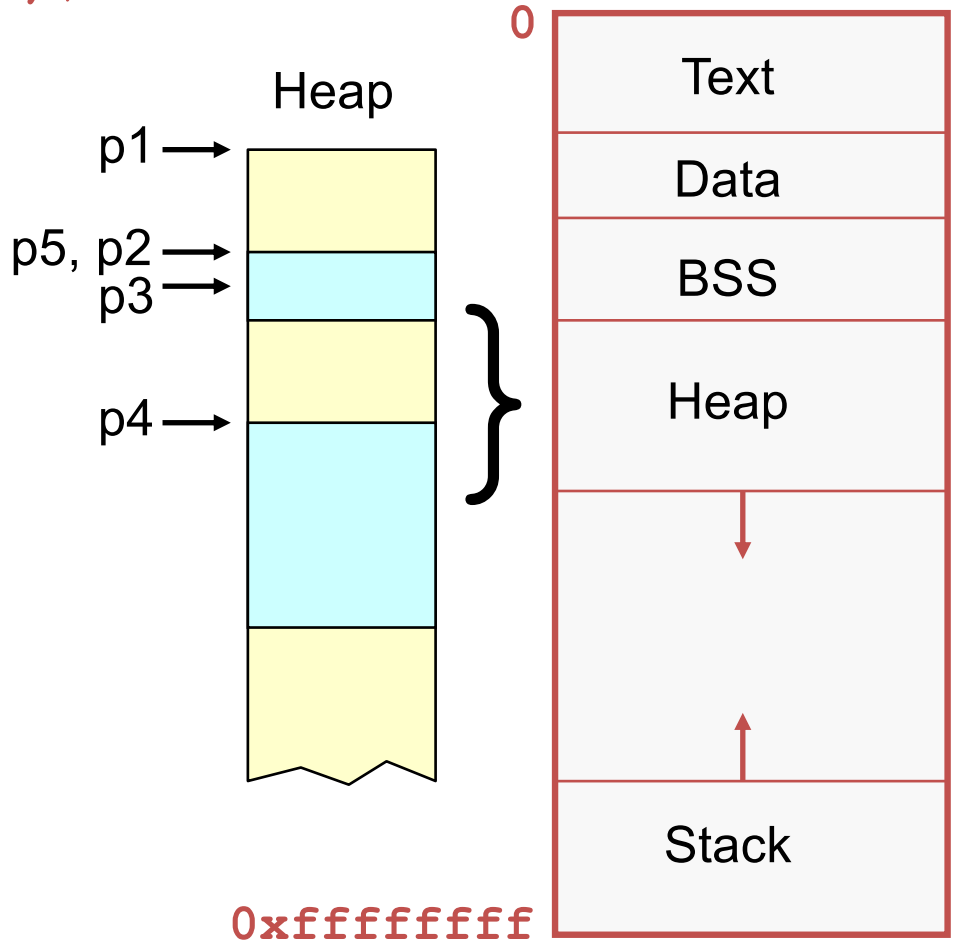
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Free List

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

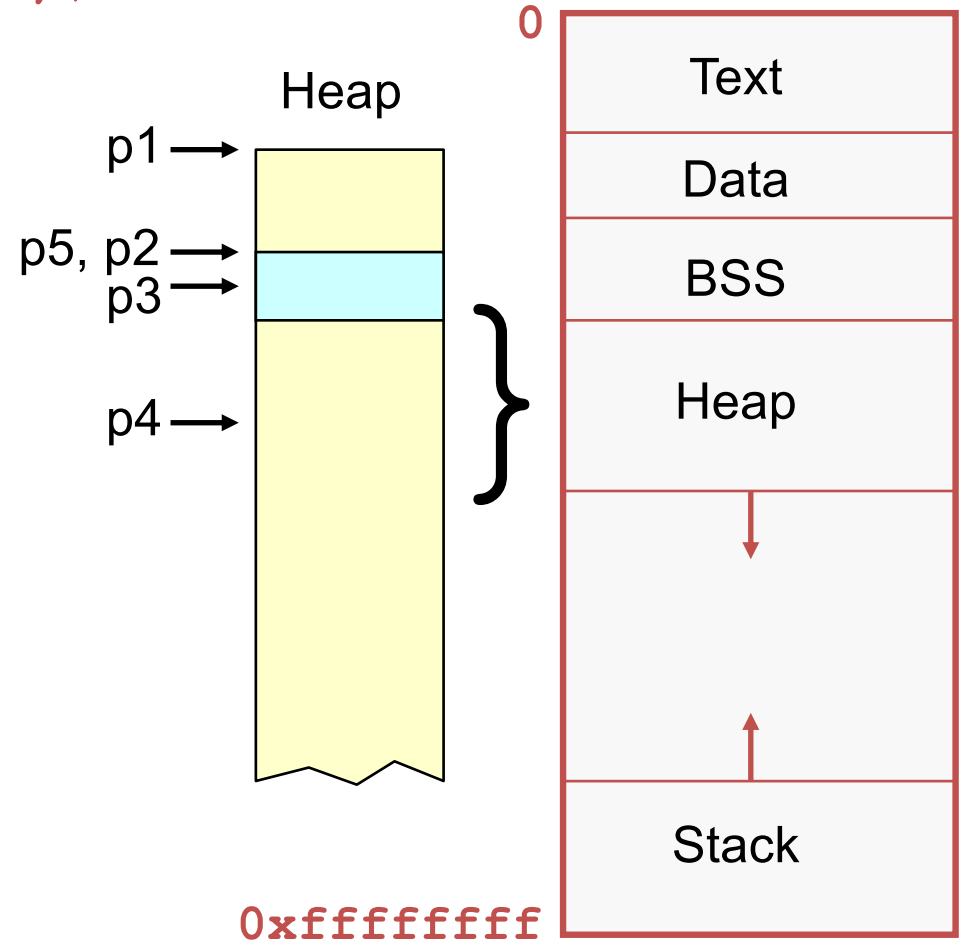
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Free List

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

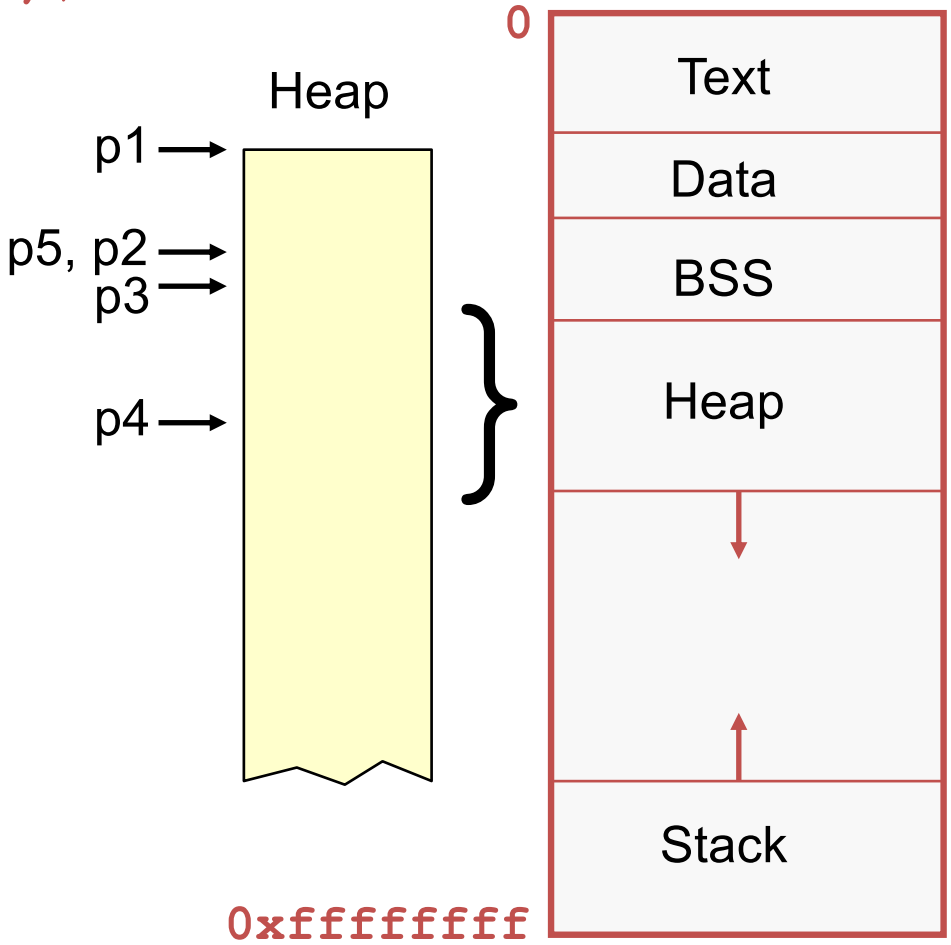
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Free List

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);

  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Free List: Text Description

- Originally free list just holds one big piece of open memory
- List of C commands and their consequences:
  - char *p1 = malloc(3)
    - 3 bytes are allocated on the heap
    - p1 holds address 100, the beginning of the 3 bytes
  - char *p2 = malloc(1)
    - 1 byte is allocated on the heap
    - p2 holds address 103 the beginning of the byte
  - char *p3 = malloc(4)
    - 4 bytes are allocated on the heap
    - p3 holds address 104 the beginning of the 4 bytes
  - free(p2)
    - deallocate the memory pointed to by p2
    - now there is a gap of free memory from addresses 103 to 104
      - note: since we are using a free list allocation scheme this memory can be reallocated later
      - free list now has 2 members on it
  - char *p4 = malloc(6)
    - look through free list for a piece of memory big enough
    - 6 bytes allocated on the heap
    - p4 holds address 108 the beginning of those 4 bytes
  - free(p3)
    - deallocate the memory pointed to by p3
    - now there is a gap of free memory from addresses 103 to 108.
      - the gap from freeing p2 is coalesced with the new gap made by freeing p3
  - char *p5 = malloc(2)
    - look through free list for a piece of memory big enough
      - see the gap from 103 to 108 and notice that it can hold 2 bytes
      - split the free memory and allocate p5 in part of it
    - 2 bytes allocated on the heap
    - p4 holds address 103 the beginning of those 2 bytes
    - now is a gap from 105 to 108
  - free(p1)
    - deallocate the memory pointed to by p1
    - now is a gap from addresses 100 to 103
      - free list now has 3 members on it
  - free(p4)
    - deallocate the memory pointed to by p4
    - gaps in memory are coalesced
      - now free list only has 2 members
        - » a gap from 100 to 103
        - » the rest of the free heap memory
  - free(p5)
    - deallocate memory pointed to by p5
    - now all of the original heap memory is free and can be reallocated as needed

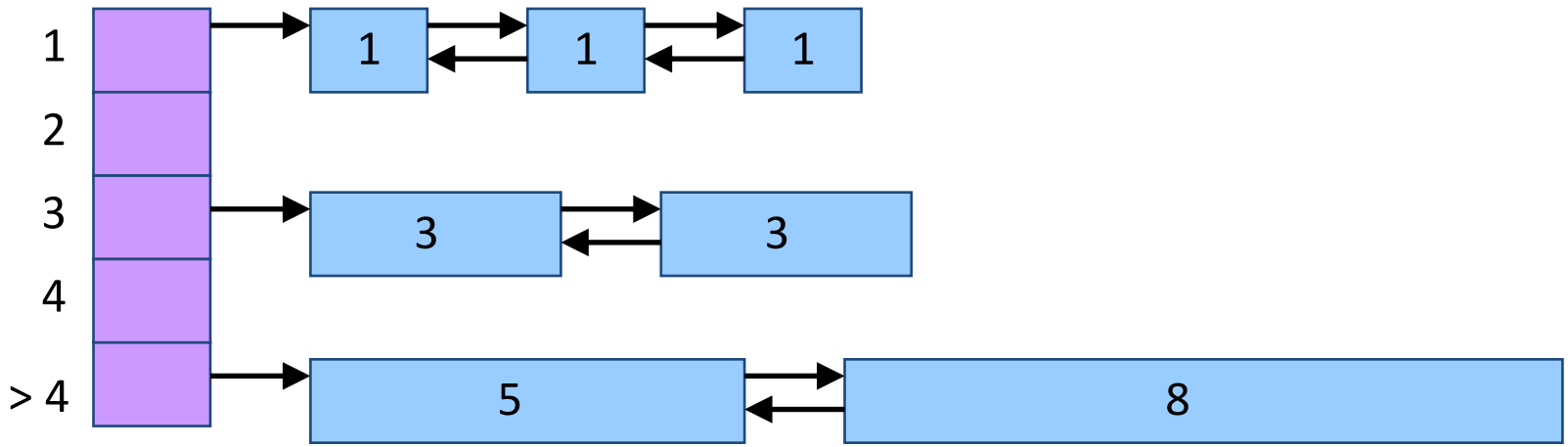
Allocation Techniques: Free List Performance

Performance

- What do we know about the performance of best-fit?
- Slow! Need to scan the free list.
- Trouble: Free chunks are different sizes
- Solution: Binning!
  - Divide list by chunk size

Binning Strategies: Exact Fit

- Have a bin for each chunk size, up to a limit
  - Advantages: no search for requests up to that size
  - Disadvantages: many bins, each storing a pointer
- Except for a final bin for all larger free chunks
  - For allocating larger amounts of memory
  - For splitting to create smaller chunks, when needed

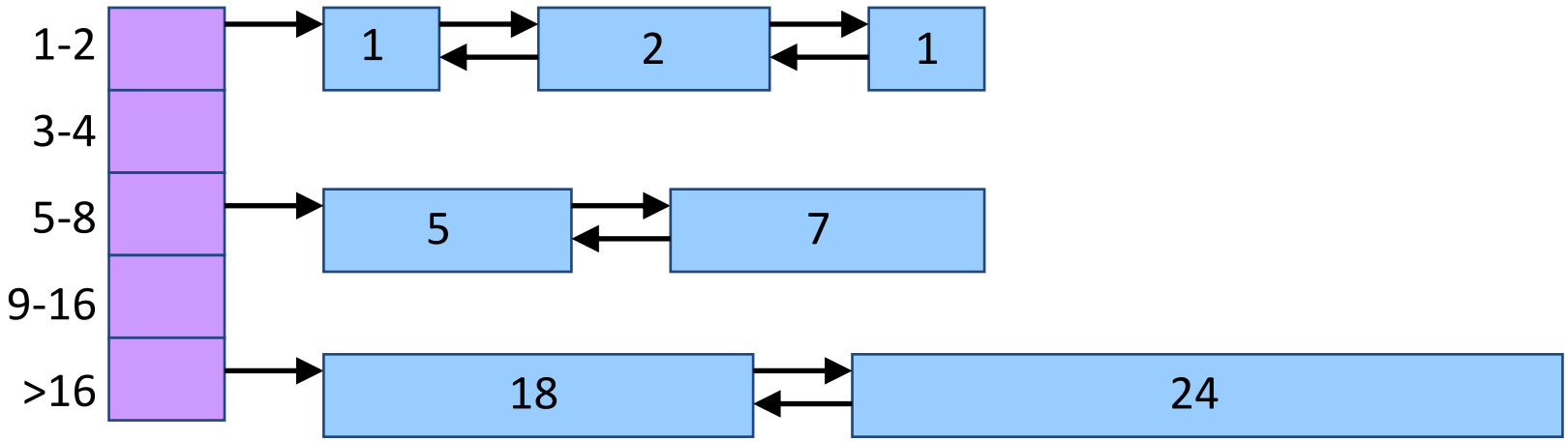


Binning Strategies: Exact Fit Text Description

- Bins are implemented with linked lists of open memory chunks
- Bin for size 1 has 3 chunks of open memory in it
- Bin for size 2 is empty
- Bin for size 3 has 2 chunks of open memory in it
- Bin for size 4 is empty
- Bin for sizes greater than 4 has 2 chunks of open memory in it
  - one chunk of size 5
  - another chunk of size 8
  - these can be taken and split up later if needed

Binning Strategies: Range

- Have a bin cover a range of sizes, up to a limit
  - Advantages: fewer bins
  - Disadvantages: need to search for a big enough chunk
- Except for a final bin for all larger free chunks
  - For allocating larger amounts of memory
  - For splitting to create smaller chunks, when needed



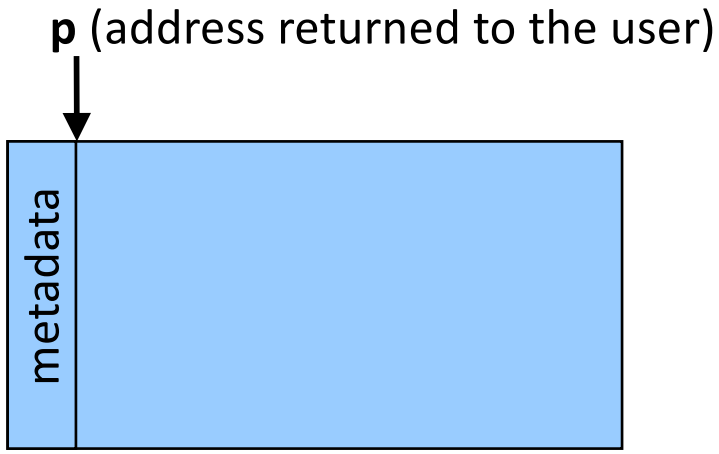
Binning Strategies: Range Text Description

- Bin for sizes 1 and 2 holds 3 chunks of open memory
  - 2 chunks of size 1 and 1 chunk of size 2
- Bin for sizes 3 and 4 is empty
- Bin for sizes 4 and 5 holds 2 chunks of open memory
  - 1 chunk of size 4 and the other of size 5
- Bin for sizes 6 and 7 is empty
- Bin for sizes greater than 7 holds 2 chunks of memory
  - One of size 10 and the other of size 14

Deallocation with Free List

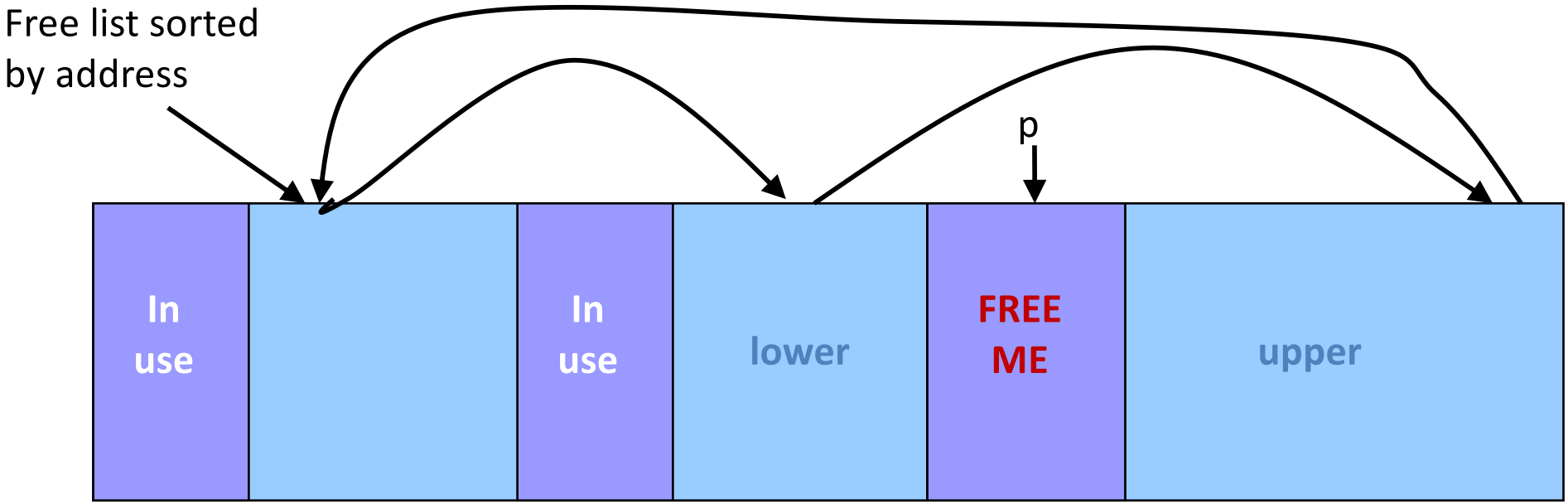
Deallocation

- Free function gets a pointer to the memory block
  - Called by user for explicit
  - Called from another algorithm for automatic
- Free function inserts block into the list
  - Identify the start of entry
  - Find the location in the free list
  - Add to the list



Coalescing With Neighbors

- Scanning the list finds the location for inserting
- Coalescing into larger free blocks
  - Check if contiguous to upper and lower neighbors



Question

What advantage does bump pointer allocation have over free-list allocation?

A. No internal fragmentation
B. No external fragmentation
C. Memory re-use
D. Fast allocation