CS 3410 Lab 3

Fall 2026

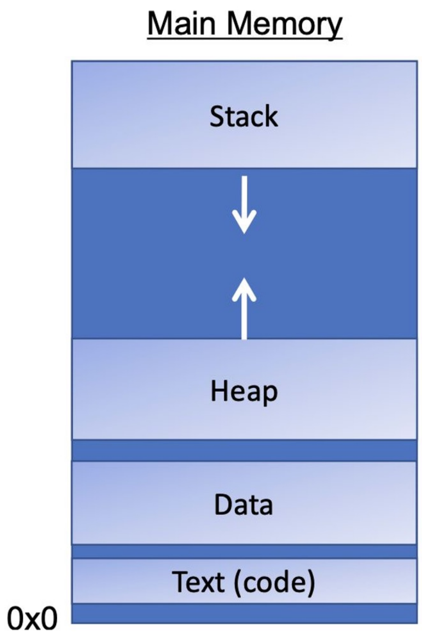
Agenda

1. C Memory Management
2. Priority Queue
3. A3 Tips

C Memory Management

The Heap

- The heap is an area of memory below the stack that grows up towards higher addresses
- Unlike the stack, where memory goes away when a function finishes, the heap provides memory that persists until the caller is done with it



How do we access memory on the heap?

- `malloc()`: Request a pointer to a contiguous block of memory on the heap
- `free()`: Release or deallocate the allocated memory back to the operating system

`malloc()` and `free()` syntax

```
int main() {
    int *ptr = (int*)malloc(sizeof(*ptr) * 10);
    // ... do some things
    free(ptr);
}
```

Rule of Thumb

Every call to `malloc()` should have a corresponding call to `free()`

- An allocation that is not freed by the time the program ends is called a **Memory Leak**

Dual Meaning of * Symbol: Type Declaration

```
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```

Type decl:
type of "a" is a pointer to an int

Dual Meaning of * Symbol: Dereference Operator

```
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```

*** is a dereference operator**

It says:
update the **memory** location pointed to by “a” with the **value** stored in “c”

Exercise: Draw out the references between memory blocks.

Note: Assume all the addresses are of 2 bytes and the integer defaults to 4 bytes.

```
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```

Stack		
Address	Name	Value
0xFFFF	c	10
0xFFFE		
0xFFFD		
0xFFFC		
0xFFFB	d	5
0xFFFA		
0xFFF9		
0xFFF8		
0xFFF7		
0xFFF6		
0xFFF5		

Heap		
0x0008		
0x0007		
0x0006		
0x0005		
0x0004		
0x0003		
0x0002		
0x0001		
0x0000		
Address	Name	Value

Exercise: Draw out the references between memory blocks.

Note: Assume all the addresses are of 2 bytes and the integer defaults to 4 bytes.

```
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```

Stack

Address	Name	Value
0xFFFF	c	10
0xFFFE		
0xFFFD		
0xFFFC		
0xFFFB	d	5
0xFFFA		
0xFFF9		
0xFFF8		
0xFFF7		
0xFFF6		
0xFFF5		

Address	Name	Value
0x0008		
0x0007		
0x0006		
0x0005		
0x0004		
0x0003		
0x0002		
0x0001		
0x0000		

Heap

Exercise: Draw out the references between memory blocks.

Note: Assume all the addresses are of 2 bytes and the integer defaults to 4 bytes.

```c
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```

Stack

Address	Name	Value
0xFFFF	c	10
0xFFFE		
0xFFFD		
0xFFFC		
0xFFFB	d	5
0xFFFA		
0xFFF9		
0xFFF8		
0xFFF7	a	0x0005
0xFFF6		
0xFFF5		
0xFFF4		

Heap

Address	Name	Value
0x0008	c copy	10
0x0007		
0x0006		
0x0005		
0x0004		
0x0003		
0x0002		
0x0001		
0x0000		

Exercise: Draw out the references between memory blocks.

Note: Assume all the addresses are of 2 bytes and the integer defaults to 4 bytes.

```c
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```

Stack

Address	Name	Value
0xFFFF	c	10
0xFFFE		
0xFFFD		
0xFFFC		
0xFFFB	d	5
0xFFFA		
0xFFF9		
0xFFF8		
0xFFF7	a	0x0005
0xFFF6		
0xFFF5		
0xFFF4		

Heap

Address	Name	Value
0x0008	c copy	10
0x0007		
0x0006		
0x0005		
0x0004		
0x0003		
0x0002		
0x0001		
0x0000		

Exercise: Draw out the references between memory blocks.

Note: Assume all the addresses are of 2 bytes and the integer defaults to 4 bytes.

```c
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```

Stack

Address	Name	Value
0xFFFF	c	10
0xFFFE		
0xFFFD		
0xFFFC		
0xFFFB	d	5
0xFFFA		
0xFFF9		
0xFFF8		
0xFFF7	a	0x0005
0xFFF6		
0xFFF5	b	0xFFF6
0xFFF4		

Heap

Address	Name	Value
0x0008	c copy	10
0x0007		
0x0006		
0x0005		
0x0004		
0x0003		
0x0002		
0x0001		
0x0000		

Exercise: Draw out the references between memory blocks.

Note: Assume all the addresses are of 2 bytes and the integer defaults to 4 bytes.

```c
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```

Stack

Address	Name	Value
0xFFFF	c	10
0xFFFE		
0xFFFD		
0xFFFC		
0xFFFB	d	5
0xFFFA		
0xFFF9		
0xFFF8		
0xFFF7	a	0x0005
0xFFF6		
0xFFF5	b	0xFFF6
0xFFF4		

0x0008	c copy	5
0x0007		
0x0006		
0x0005		
0x0004		
0x0003		
0x0002		
0x0001		
0x0000		
Address	Name	Value

Heap

Exercise: Draw out the references between memory blocks.

Note: Assume all the addresses are of 2 bytes and the integer defaults to 4 bytes.

```
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```

Stack

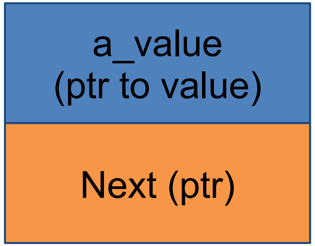
Address	Name	Value
0xFFFF	c	10
0xFFFE		
0xFFFD		
0xFFFC		
0xFFFB	d	5
0xFFFA		
0xFFF9		
0xFFF8		
0xFFF7	a	
0xFFF6		
0xFFF5	b	0xFFF6
0xFFF4		

Heap

Address	Name	Value
0x0008	~~c copy~~	~~5~~
0x0007		
0x0006		
0x0005		
0x0004		
0x0003		
0x0002		
0x0001		
0x0000		

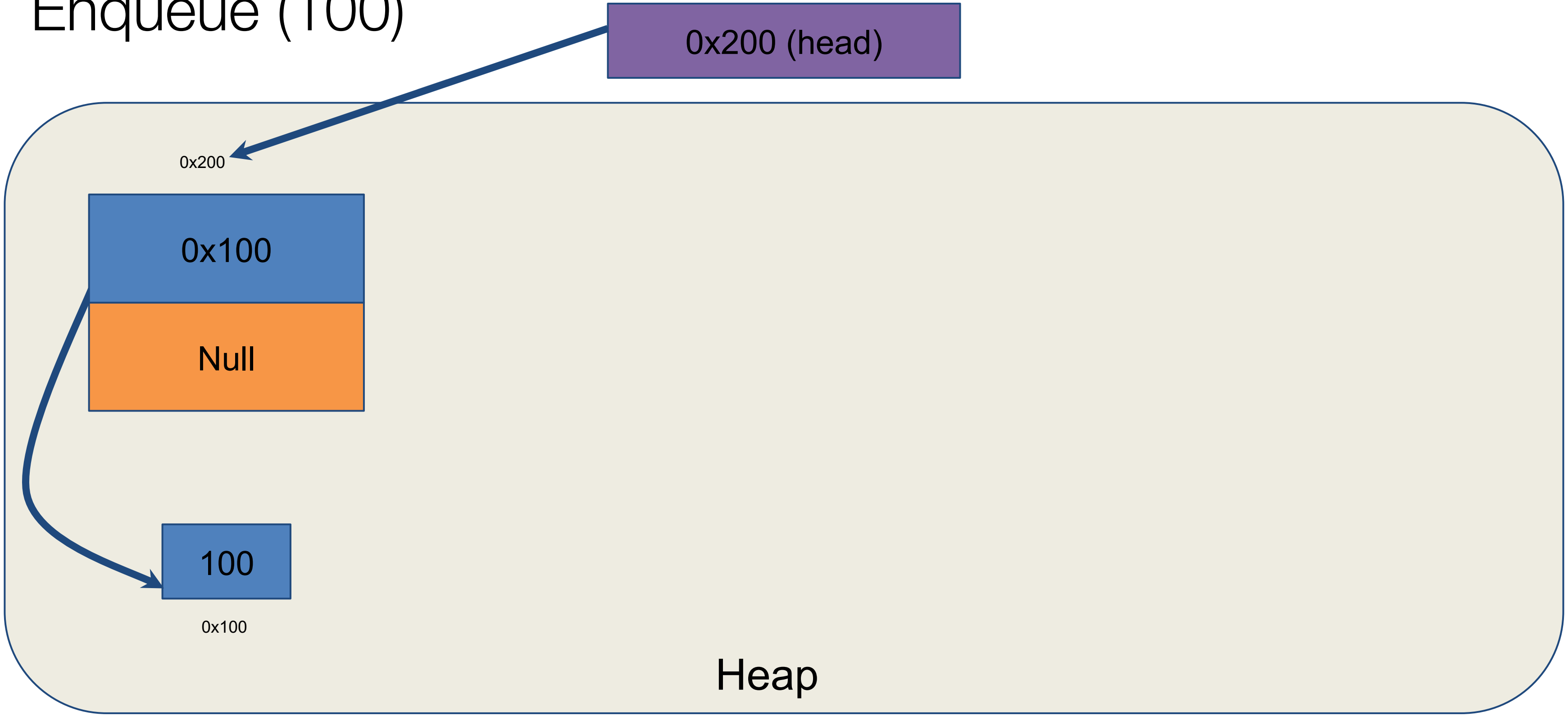
Lab: Priority Queue

Malloc PQNode

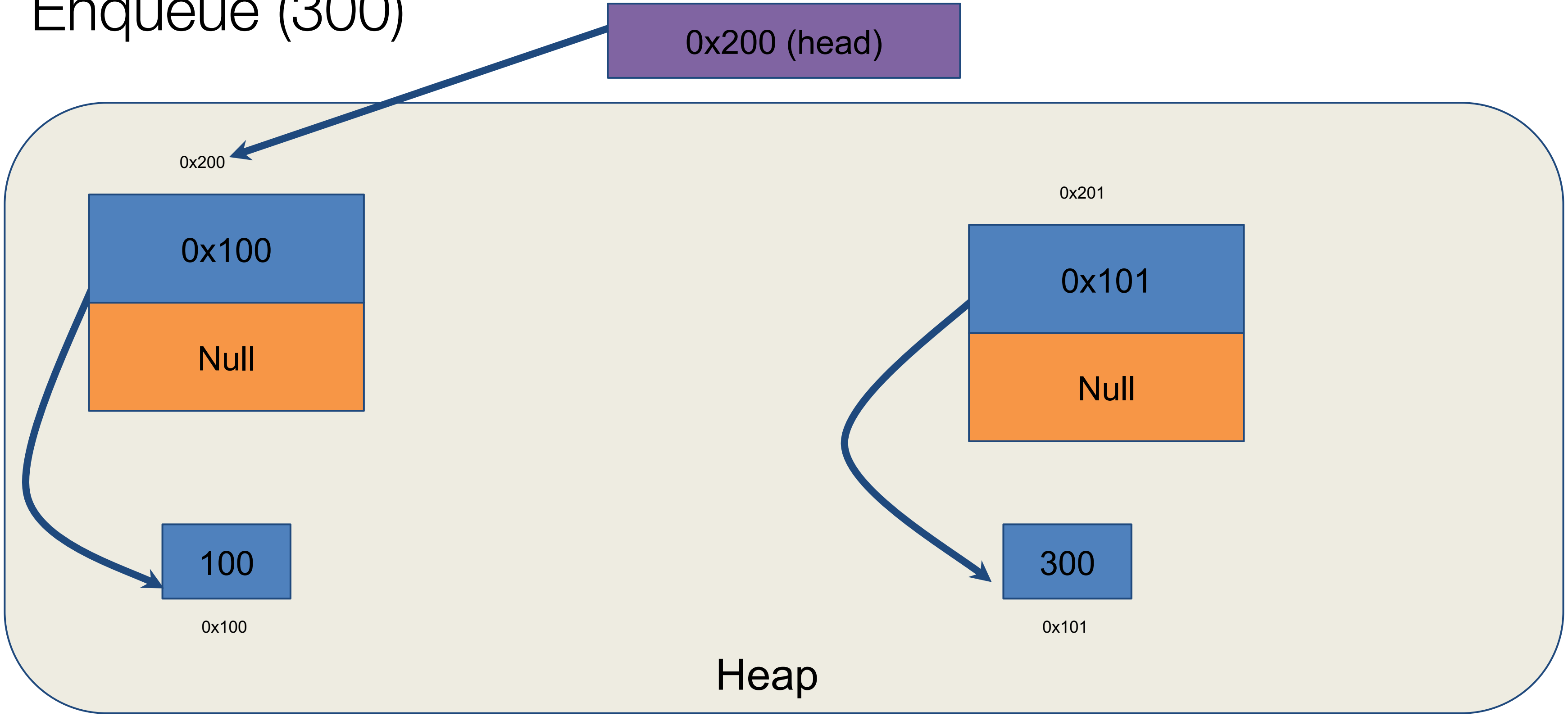


Malloc PQNode

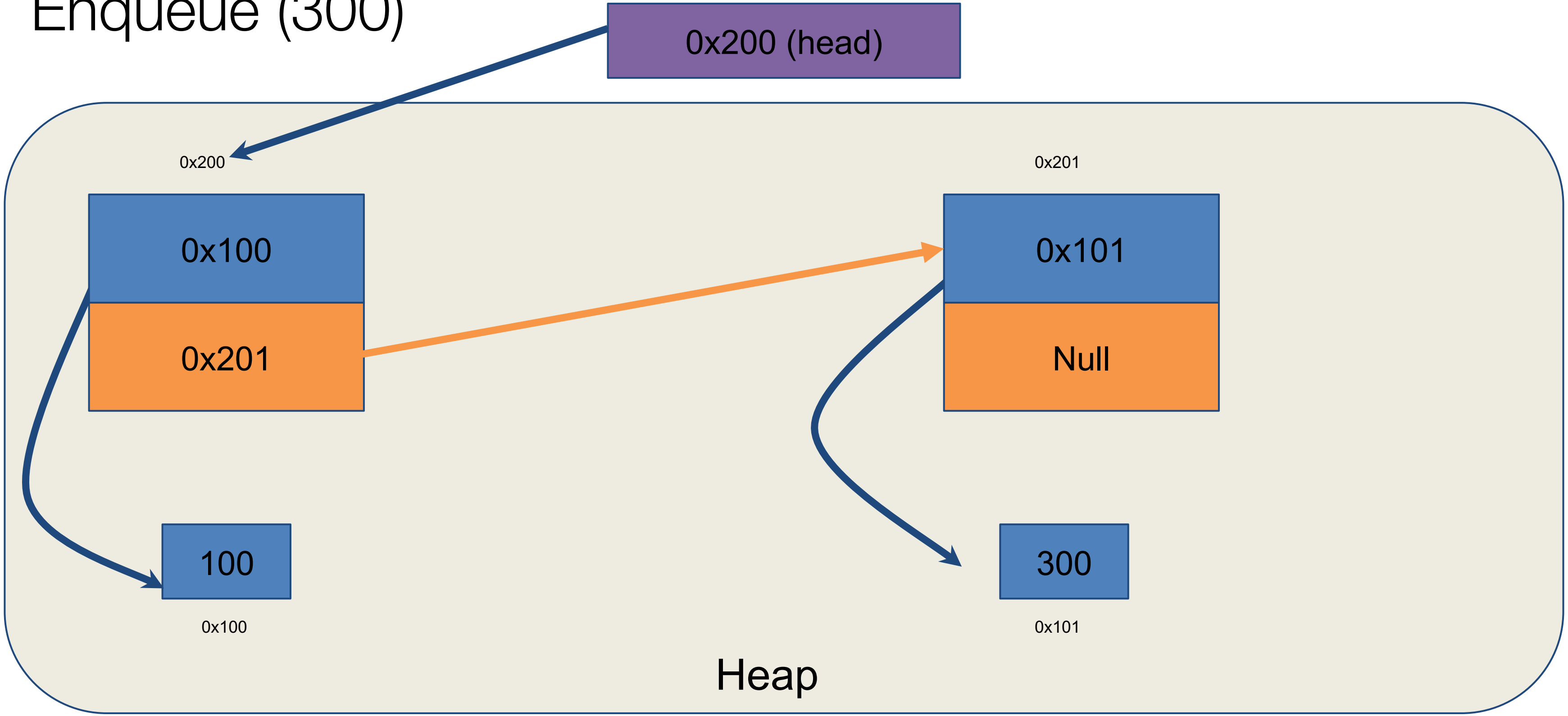
Enqueue (100)



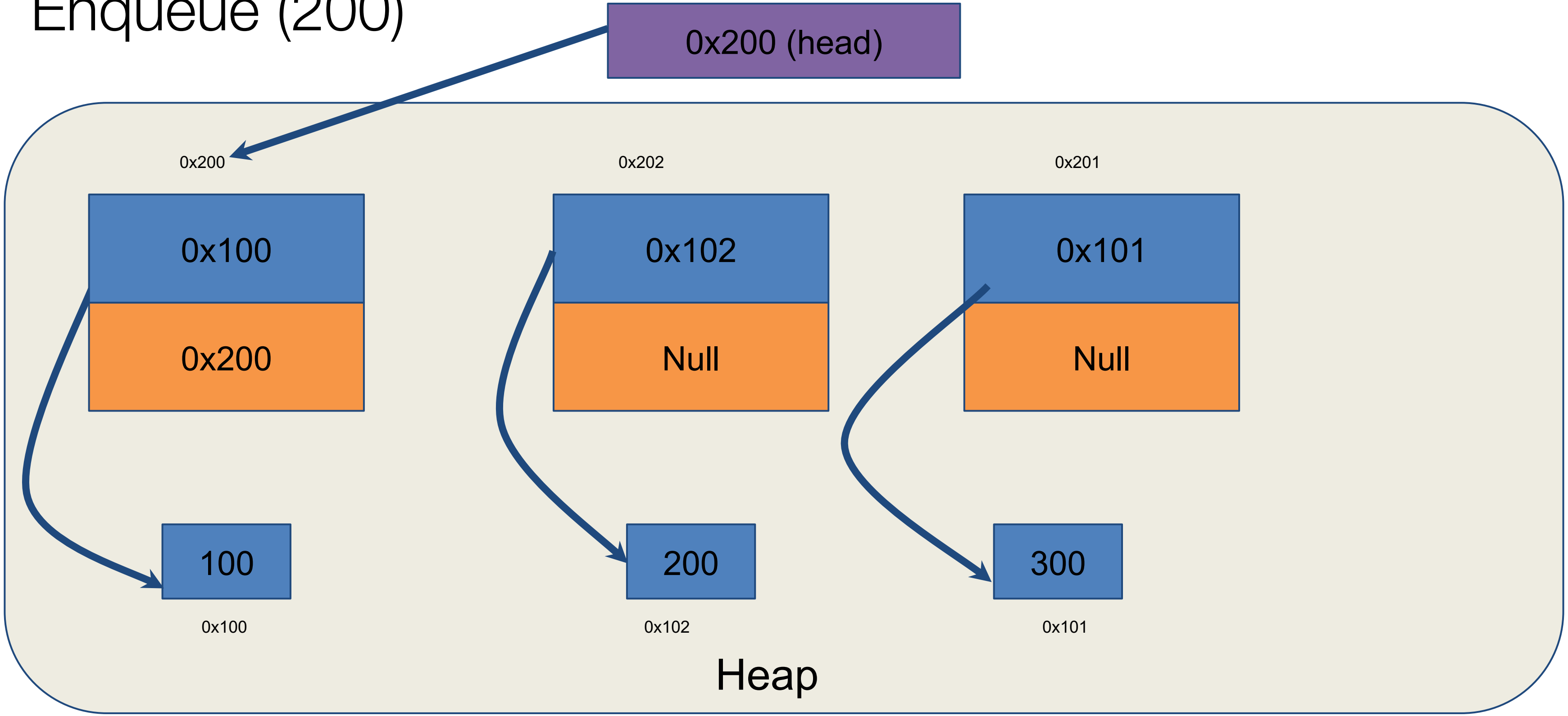
Enqueue (300)


0x200 (head)
0x200
0x100
Null
0x201
0x101
Null
100
300
0x100
0x101
Heap

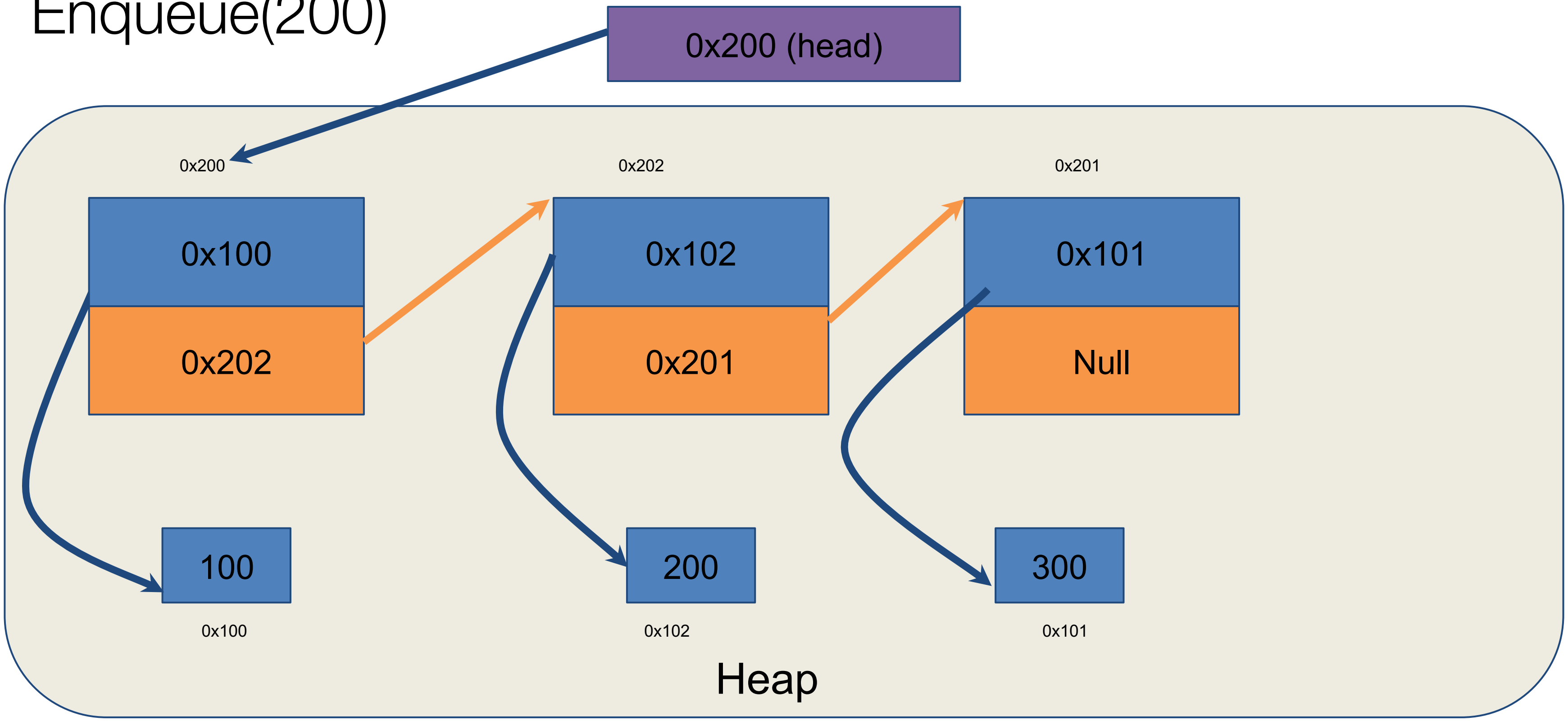
Enqueue (300)


0x200 (head)
0x200
0x100
0x201
0x201
0x101
Null
100
0x100
300
0x101
Heap

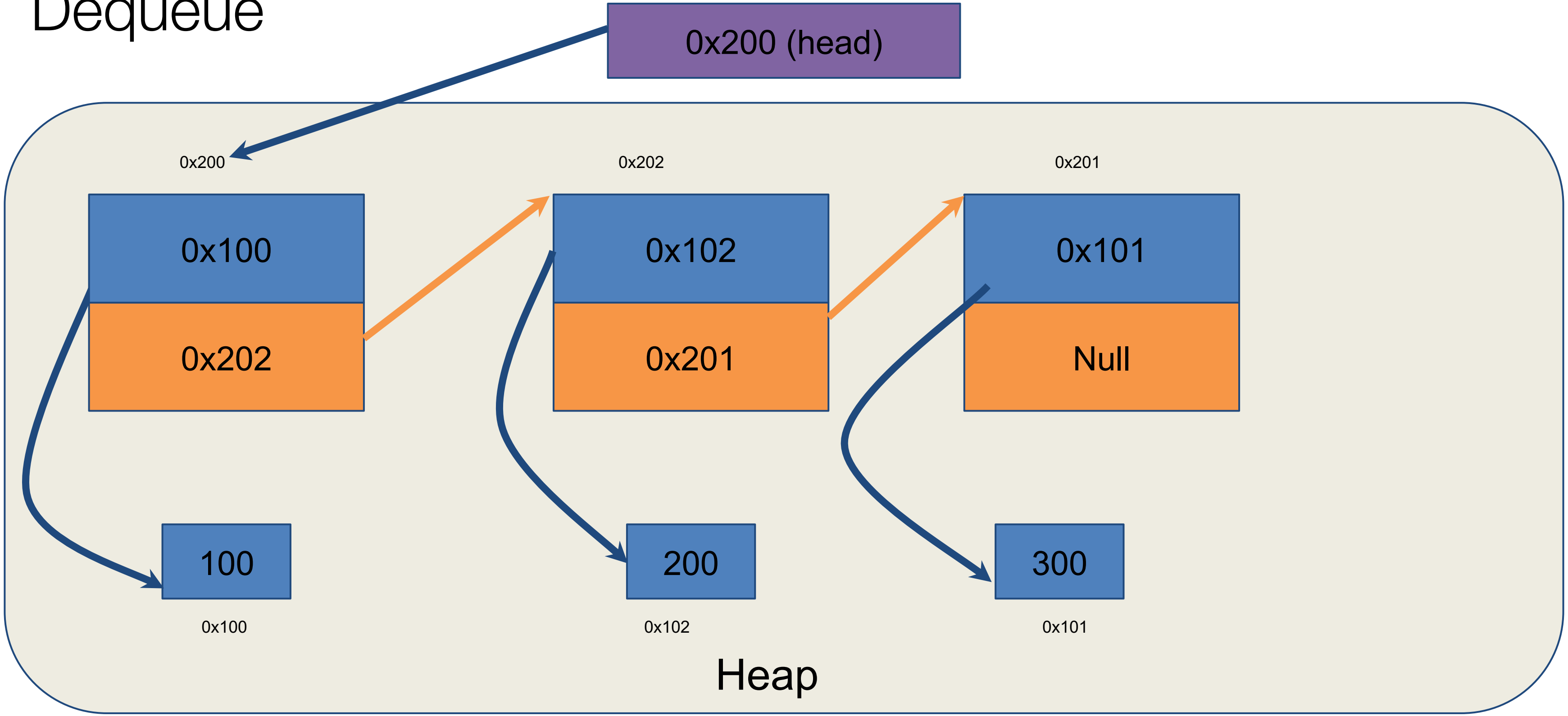
Enqueue (200)


0x200 (head)
0x200
0x202
0x201
0x100
0x200
0x102
Null
0x101
Null
100
200
300
0x100
0x102
0x101
Heap

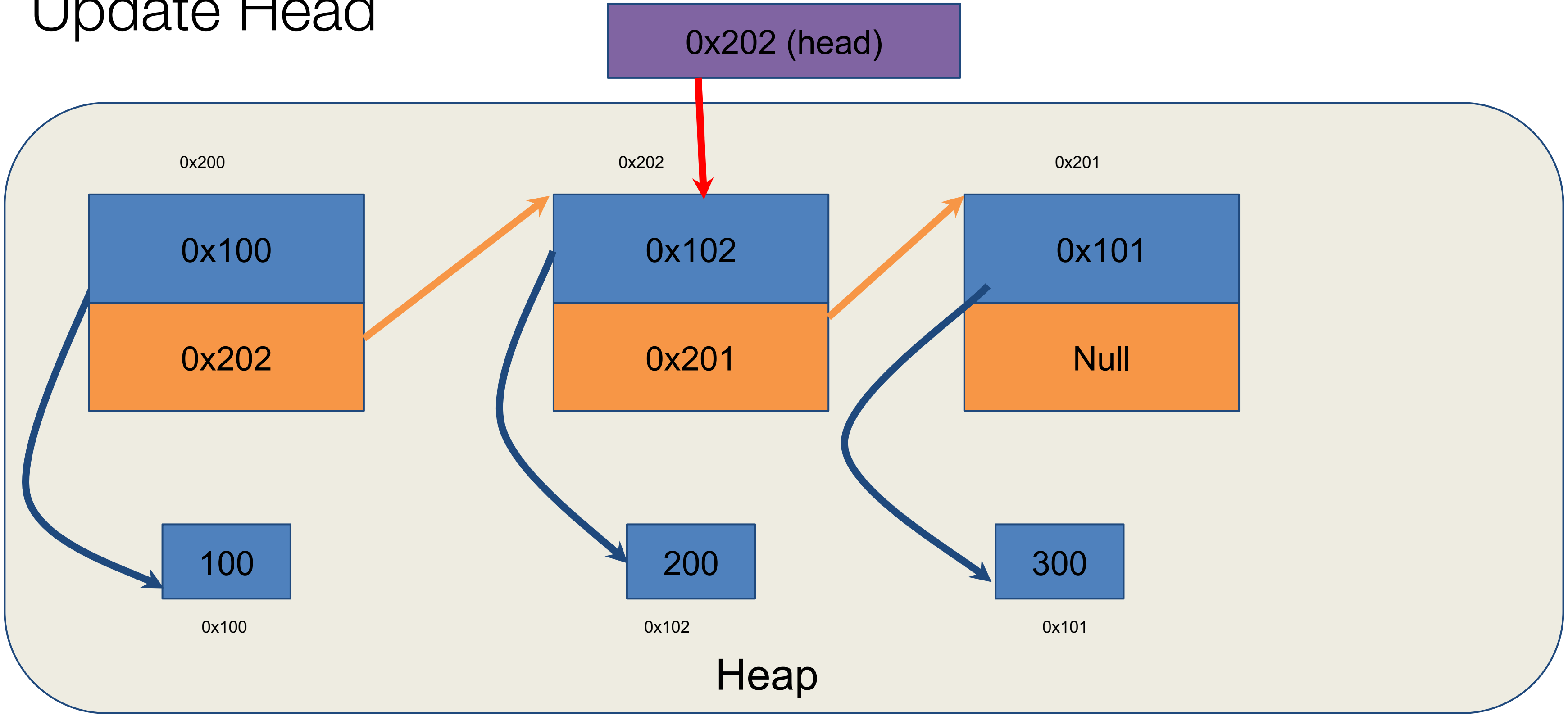
Enqueue(200)


0x200 (head)
0x200
0x202
0x201
0x100
0x202
0x102
0x201
0x101
Null
100
200
300
0x100
0x102
0x101
Heap

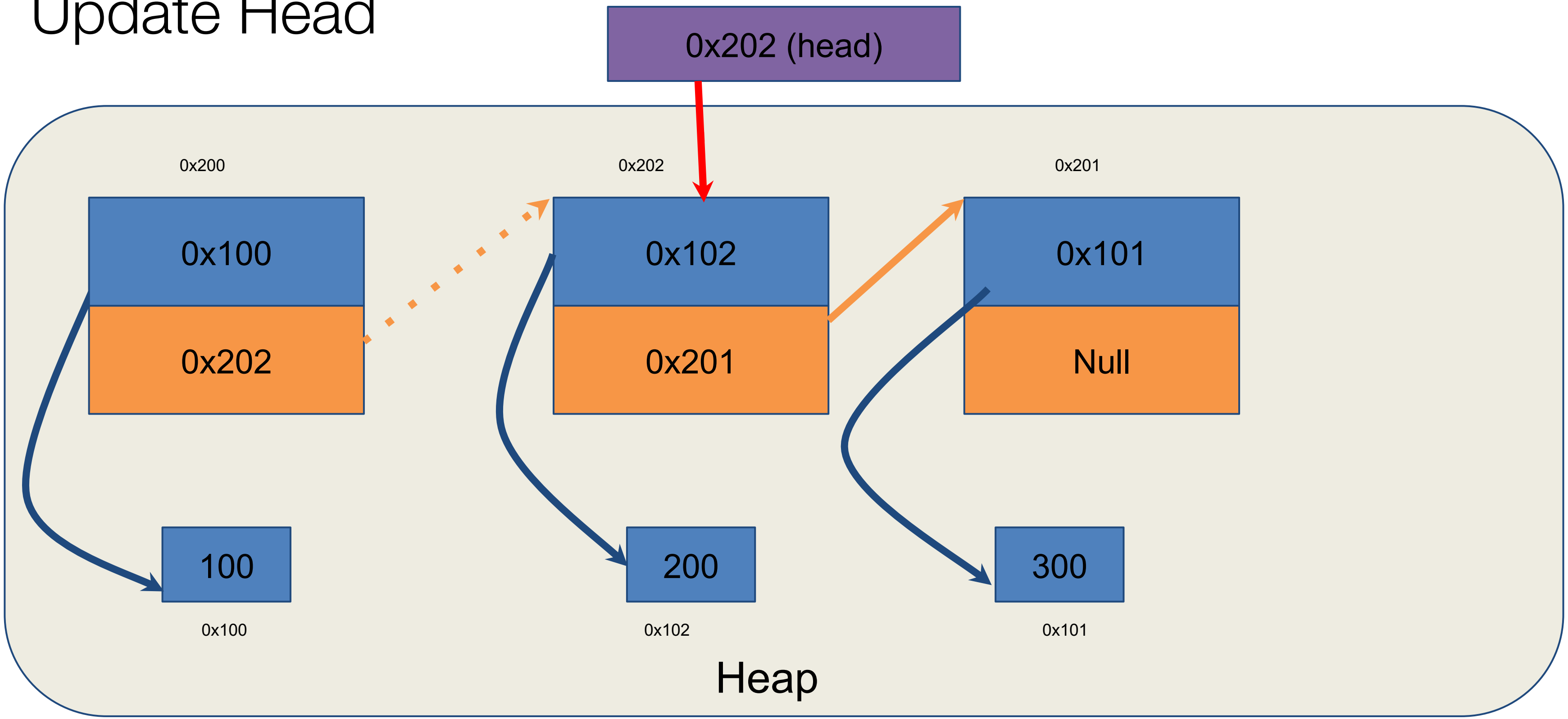
Dequeue


0x200 (head)
0x200
0x202
0x201
0x100
0x202
0x102
0x201
0x101
Null
100
200
300
0x100
0x102
0x101
Heap

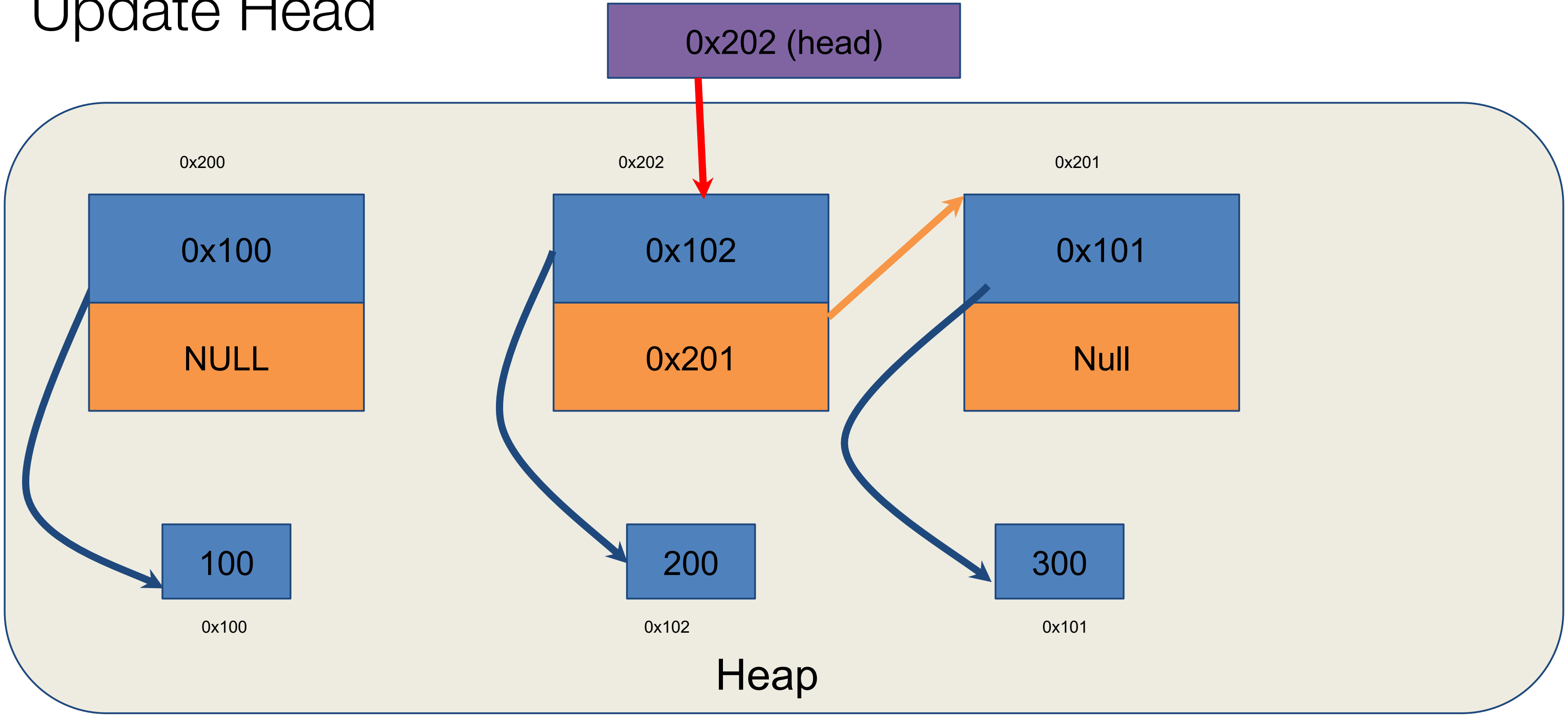
Update Head


0x202 (head)
0x200
0x202
0x201
0x100
0x202
0x102
0x201
0x101
Null
100
200
300
0x100
0x102
0x101
Heap

Update Head


0x202 (head)
0x200
0x100
0x202
0x202
0x102
0x201
0x201
0x101
Null
100
200
300
0x100
0x102
0x101
Heap

Update Head


0x202 (head)
0x200
0x100
NULL
0x202
0x102
0x201
0x201
0x101
Null
100
200
300
0x100
0x102
0x101
Heap

Lab Hints

- You can implement `stack_push` by calling pq_enqueue and passing in NULL as the compare function. If you do this, you'll need to think carefully about your implementation for `pq_enqueue`.
- You'll also need to write a compare function to order the nodes in your priority queue. Nodes are sorted in *ascending* order, first by frequency, then by ASCII value. Follow this convention when implementing the function:
  - `cmp_fn(a, b) < 0` –> a is ordered before b
  - `cmp_fn(a, b) >= 0` –> a is ordered after b

A3 Tips

Start early, start early, start early!

- This assignment has many intricacies. To give yourself enough time to test the functionality end-to-end, you'll want to start early.
- **Test thoroughly!** We've provided a unit-testing framework for you to use with test files already started for you. Add more tests as you need them to make sure your code is correct.
  - This applies to `huffman.c` as the complete test suite for Priority Queue is already given

Test your code!

- Testing before you get to Task 2 will be crucial. We've provided you with a simple unit-testing library called `cu_unit`.
- The structure of a unit test is as follows:

```
int _test_equality() {
    cu_start(); // We must call this at the start of every test
  //---------------
    int x = 4;
    int a_x = &x;
    cu_check(x == *a_x); // We can call cu_check() as many times as we want
within a test
  // ---------------
    cu_end(); // We must call this at the end of every test
}
```

Test your code!

- We can run the test by adding it to `main()`:

```
int _test_equality() {
    cu_start(); // We must call this at the start of every test
  //---------------
    int x = 4;
    int a_x = &x;
    cu_check(x == *a_x); // We can call cu_check() as many times as we want
within a test
  // --------------
    cu_end(); // We must call this at the end of every test
}

int main() {
    cu_run(_test_equality); // Run the test
    return 0;
}
```

Test your code!

- Tests will not be graded for this assignment, but it’s a good idea to test your code before moving on to the next task, as we’ll be grading your code for each section individually.
- Test your priority queue with different types, comparison functions, etc.
- Test your Huffman tree on different files (C programs, for instance), and with a variety of characters.

Remember!

- **In A3 (not this lab), you must** use our wrapper functions `my_malloc()` and `my_free()` for any allocations or deallocations.
- They follow the exact same syntax as `malloc()` and `free()` and perform the same kind of allocation/deallocation.
- The only thing extra is that they log the operation, which will help you and us detect and pinpoint memory leaks in your implementation.

Good luck!