Heap Arrays and Linked Lists

Steven R. Bagley

Recap

- Data is stored in variables
- Can be accessed by the variable name
- Or in an array, accessed by name and index
- Variables and arrays have a type
- Create our own data `structures`
- Allocate our own memory

Allocating memory

- `int *p = (int *)malloc(sizeof(int));`
- Find the size of a `int`
- `malloc` that many bytes
- Cast the `void *` pointer to a `int *`
- And store in `p`

This generally safe, although handle with care

Always think why am I casting this?

Don’t cast just to make the errors and warnings go away

Using allocated memory

- Can use this allocated `int` like any other pointer to an `int`
- But *have* to access it via the pointer
- Cannot obtain a variable name for it
- Can also allocate `structs` in this fashion

`free()`

- When we've finished with allocated memory, we need to `free()` it
- Otherwise, it can't be used for anything else until the program quits
- But can't free it until we've finished with it (the program has no more pointers with it)
- Otherwise, accessing it will cause problem

`free()`

- `void free(void *ptr)`
- Pass it the pointer to the memory you want freeing
- Must be allocated via `malloc()`
- Afterwards, the variable containing the pointer should be cleared as it is no longer valid

Allocating `struct`

- `struct point *p = (struct point *) malloc(sizeof(struct point));`
- Find the size of a `struct point`
- `malloc` that many bytes
- Cast the `void *` pointer to a `struct point *`
- And store in `p`

This generally safe, although handle with care
Always think why am I casting this?
Don’t cast just to make the errors and warnings go away

Heap `structs`

- Can use these allocated blocks just like anything else we've pointed at

```
p->x = 42.0;
p->y = 36.0;
struct point pt = *p;
```

- Allocating `struct`s on the heap is used a lot
- Most data is stored on the heap

Arrays on the Heap

- Arrays are represented by a pointer to the base of the array
- Each element in the array is laid out in memory sequentially
- The array operator `[]` works just as well on a pointer as on an array

Arrays on the Heap

- If we can `malloc` space for one thing
- Then we can `malloc` space for two things
- Or three, or four etc…
- Treat that pointer as the base of the array

Arrays on the Heap

- Find out how big one thing is using `sizeof()`
- If we want to allocate space for `n` things
- Multiple `sizeof()` by `n`
- Gives us the number of bytes to allocate
- So an array of `42 ints`:

```
int *p = malloc(42 * sizeof(int));
```

Solving our pictures

- Don’t decide array size at compile-time
- Create it at run-time using `malloc()`
- But how big should it be?
- Four options…

Size of Path Array

- Option One — read through the file twice
- First time through, count lines until `stop`
- Then allocate memory using `malloc`
- Then use `fseek` to go back to the beginning and read again
- Problem — might be reading from something that can't be seeked (e.g `stdin`)

Size of Path Array

- Option Two — Cheat…
- Rather than putting stop at the end
- Make the first line contain the number of elements
- Read that
- Use value to allocate the array

Size of Path Array

- Option Three — copy data if array gets full
- Allocate the array of a certain size
- If we fill it, allocate a new larger array
- And copy the data over using `memcpy`
- Works, but can leave memory fragmented and slows program while copying

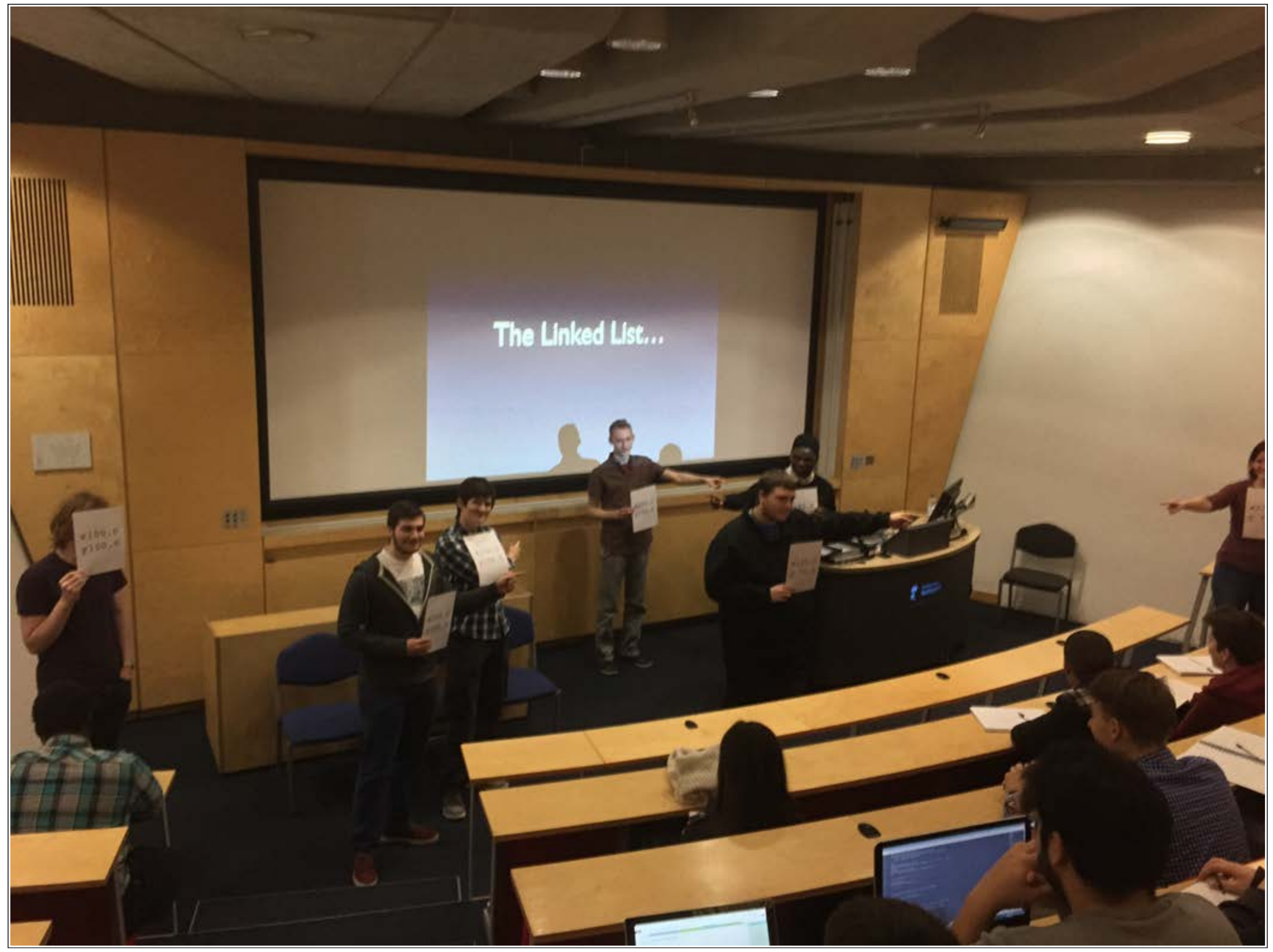
Size of Path Array

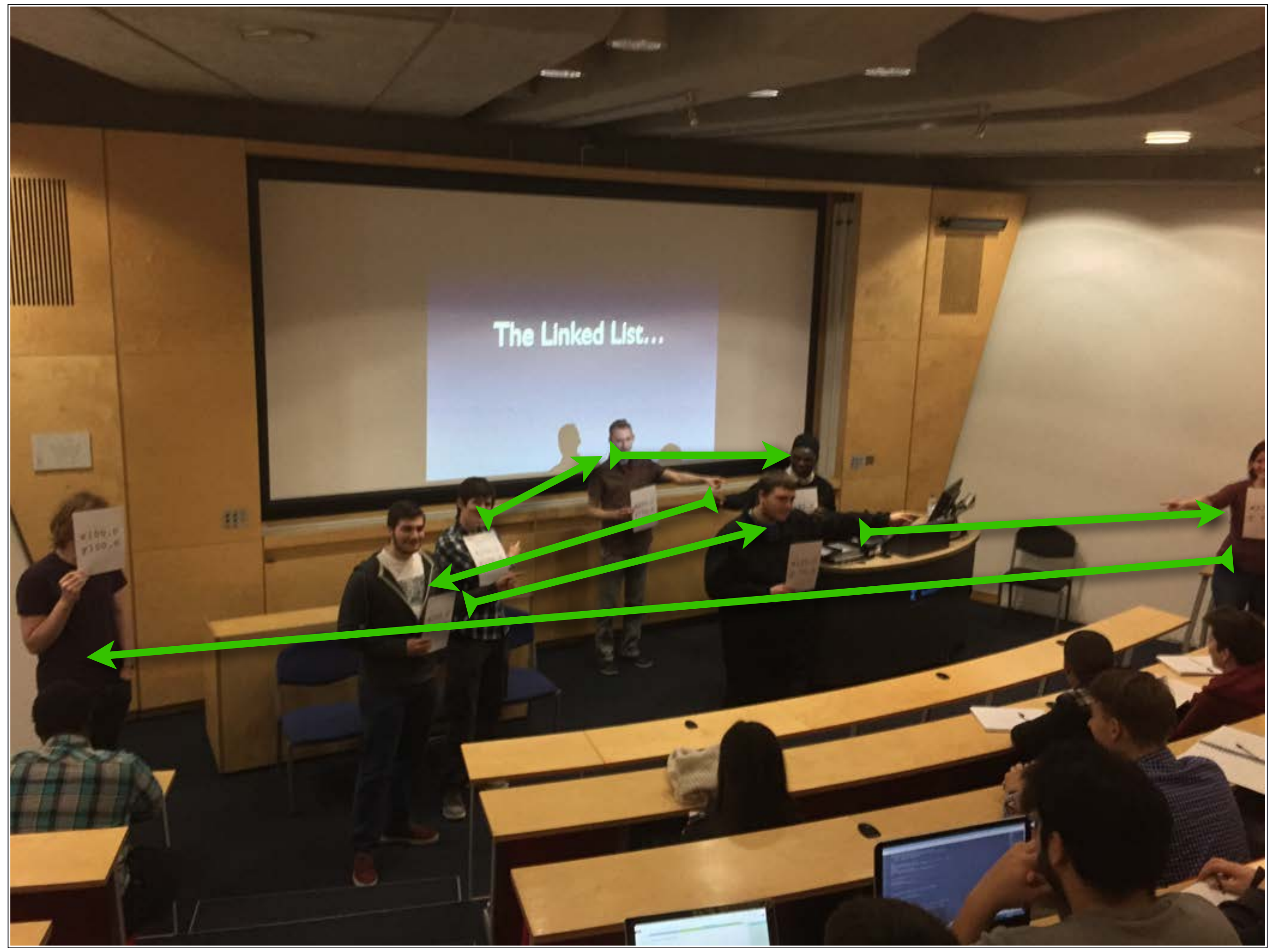
- Option Four — Don’t use an Array
- The problem we have is that an array’s size is fixed when created
- If it needs to grow, we need to create a new one and copy the data
- So use a different data structure that doesn’t have this limitation, such as…

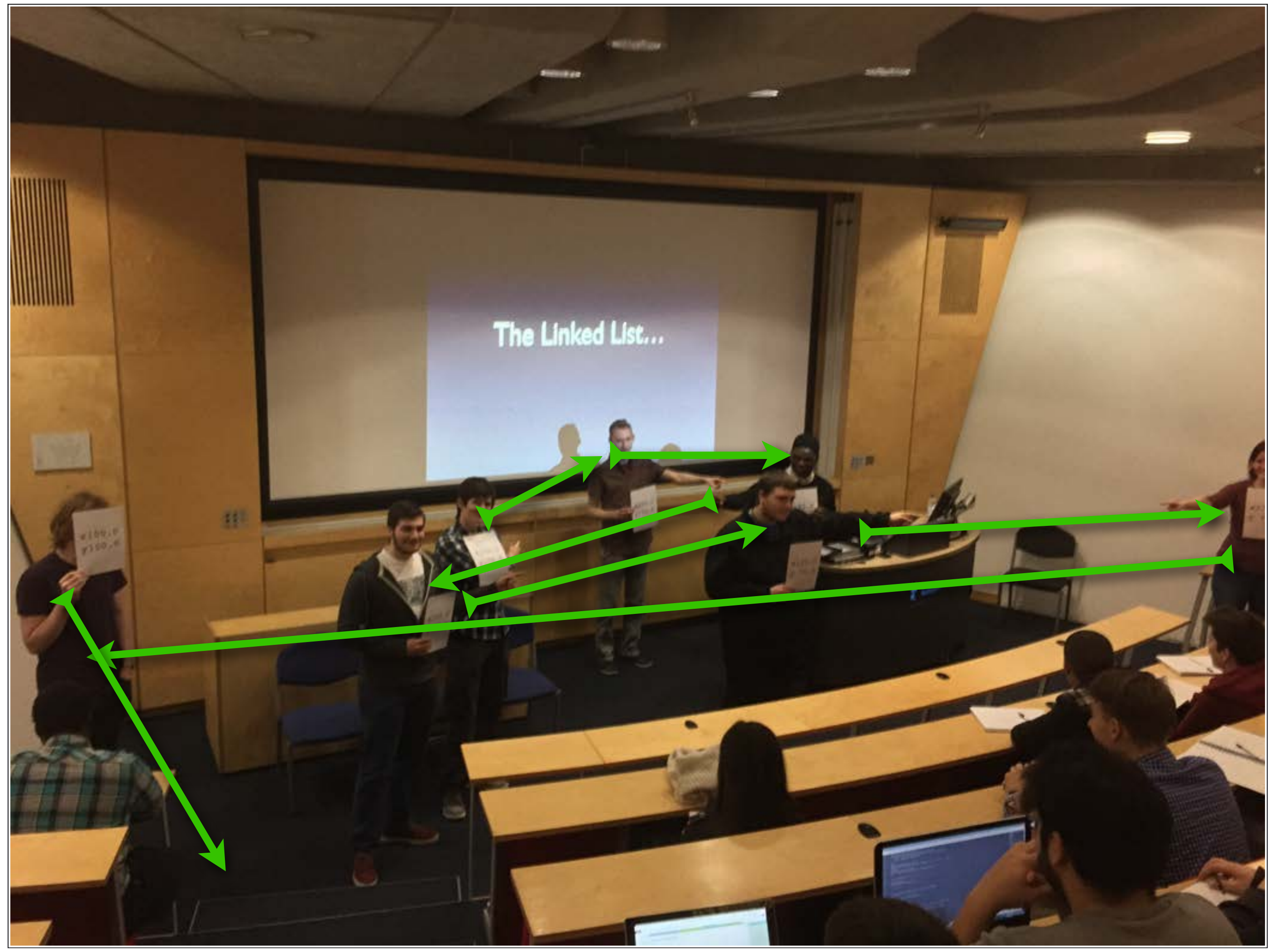
SET / SEM

- As in G51CSA, you get a chance to give feedback on the module
- Be careful with the scale…
- Done online at http://bluecastle.nottingham.ac.uk

The Linked List…


The Linked List...


The Linked List...


The Linked List...

Linked List

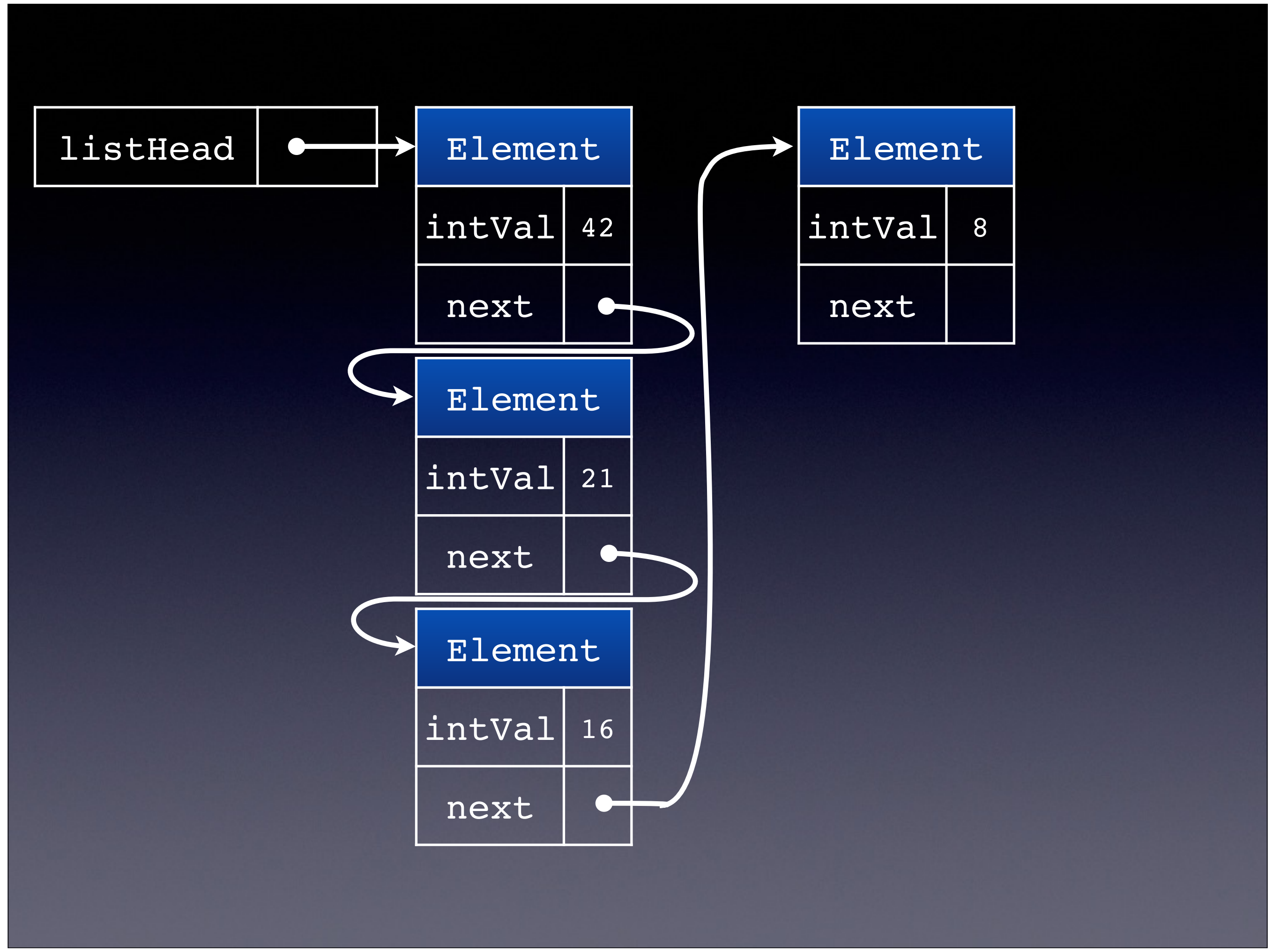
- Ordered collection
- Size not fixed
- Grows and shrinks as elements are added
- Add/remove elements from the list easily
- Trickier to access a particular element in the list (sequential access rather than random access)

Linked List vs Array

Linked List	Array
Ordered	
Grows/shrinks on demand	Fixed size
Easily add/remove elements	Difficult to add/remove elements
Sequential access	Random Access

Linked List

- Linked lists are built using `structs`
- These `structs` store the data as variables inside them just as the others we've seen
- But they also have another variable
- A pointer to the *next* item in the list (sometimes called the *link*)
- Last item points at nothing (i.e. `NULL`)



Last struct, doesn't point at anything (null) to signify the end of the list

Linked List

- Have to use pointers
- Have to allocate the memory for the `struct`s using `malloc`
- Store pointer to the first `struct` (at least)
- Easy to access first element (we have a pointer to it)

If we didn’t use pointers the struct would fill up the computers memory

Accessing an element

- To access the n^{th} element
- Start at the head of the list
- This refers to the first element
- Follow the link to the next element
- This is the second element
- And so on until you reach the n^{th} element

Linked List of `points`

- How could we make a linked list out of our `point`s?
- Still need the variables in the `struct` as we had before
- Add a new variable — a pointer to a `struct point`

Linked List of `points`

```
struct point
{
    float x;
    float y;
    struct point *next;
};
```

Linked List of `points`

- Allocate memory for this `struct` using `malloc`
- Set data in the normal way
- Also set `next` to `null`
- Store pointer in a variable that points to the first item
- Nothing unusual

Creating a Second Point

- Allocate memory this `struct` using `malloc`
- Set data in the normal way
- Also set `next` to `null`
- Where do we store this pointer?
- In the `next` variable of the first `point`

General case

- Unless it is the first item (`point` in this case)
- We have to know where the first item is
- The pointer to an item goes in the previous item's `next` variable
- Can then find any item by following the links from the first item

Adding to the end

- Often need to add a `struct` to the end of the list
- Three stage process
- First, allocate space for the new `struct`
- Second, find the last `struct` in the list
- Third, set the last `struct`’s `next` to point to the new `struct`

Last struct is the one where next is equal to NULL

Allocating struct

- Seen this already — use `malloc()`
- Must be allocated on the heap, no other way to do it
- Set the variables
- Set `next` to be `NULL`

Finding the last element

- Relatively straightforward
- Start at the first item
- Follow the `next` pointers until we find a struct where `next` is `NULL`
- Special case when list is empty (`head` is `NULL`)

General case

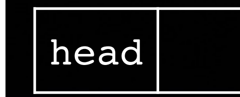
- When `head` is not `NULL`:
  - Set a pointer, `p`, to equal `head`
  - If `p->next` is `NULL`, stop as end found
  - Otherwise, set `p` to equal `p->next` (moves `p` to point to the next thing in list)
  - Repeat until end found

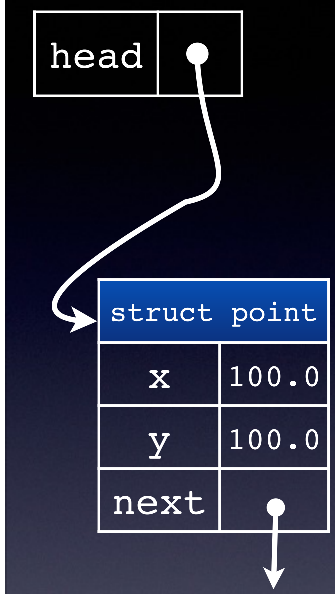
General case

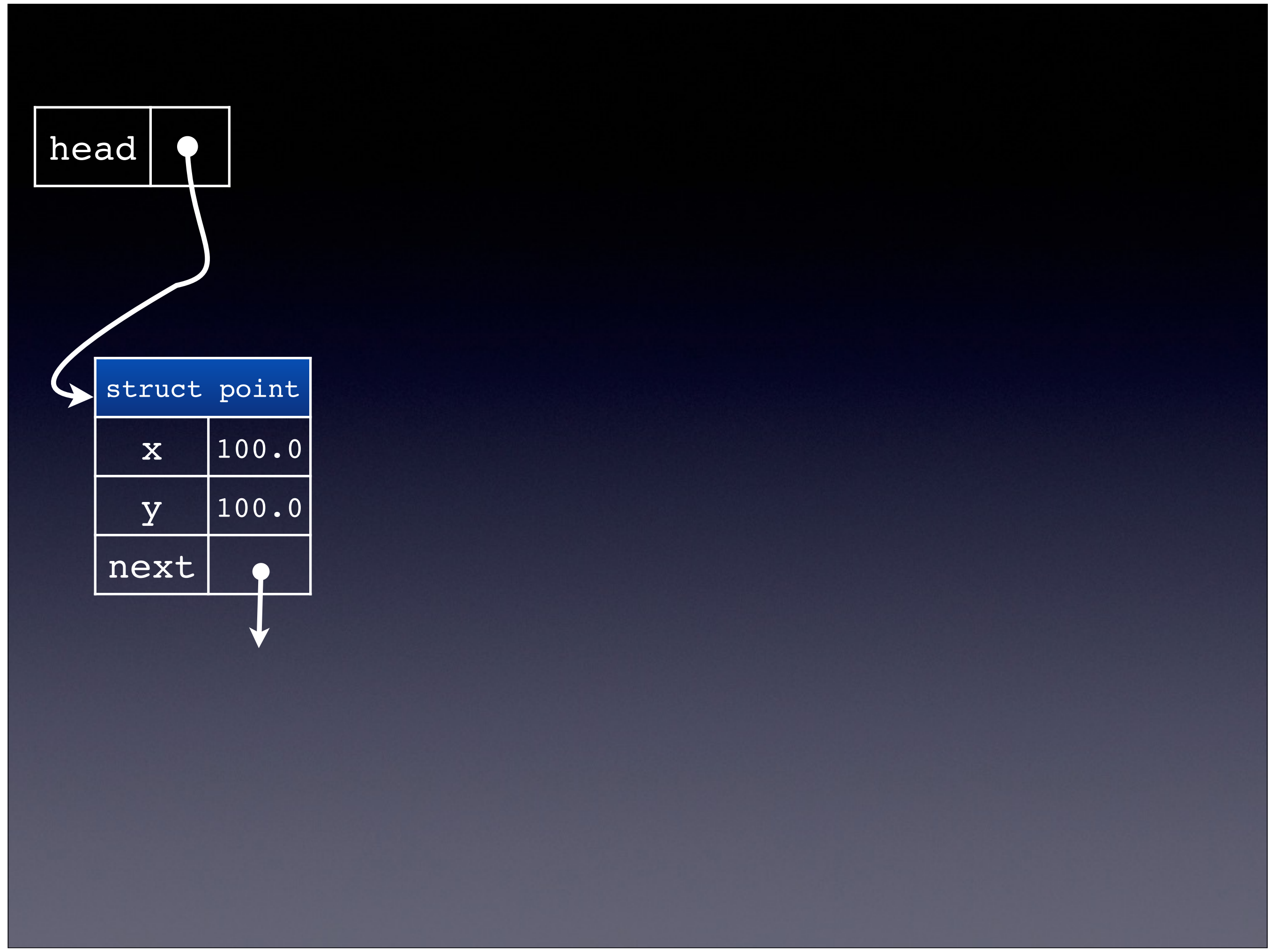
- When we reach the termination case, `p` will be pointing at the last item in the list
- Because we test whether `p->next` is `NULL` and not `p` itself
- Can then set `p->next` to point to our newly allocated `struct`

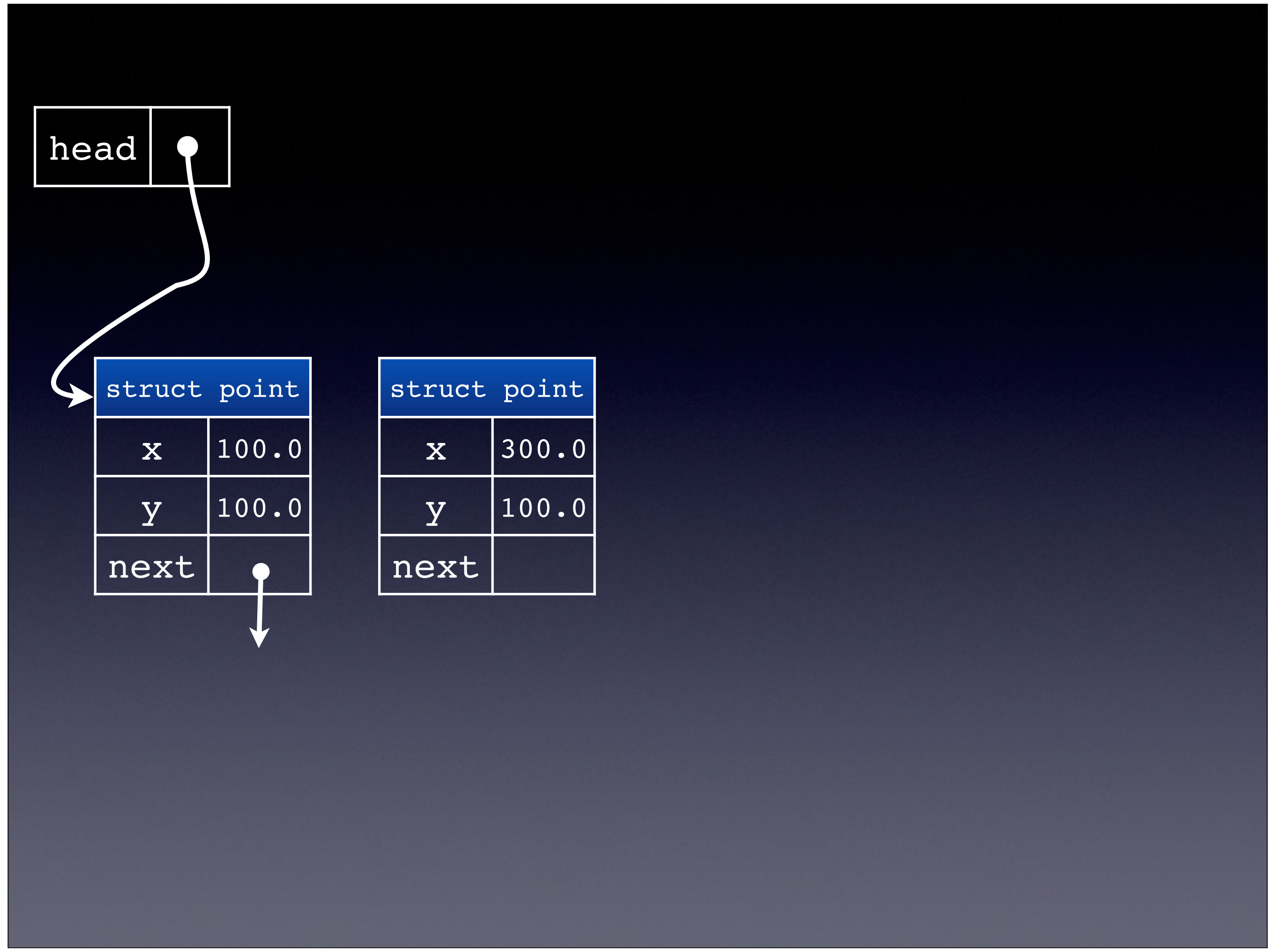
Special case

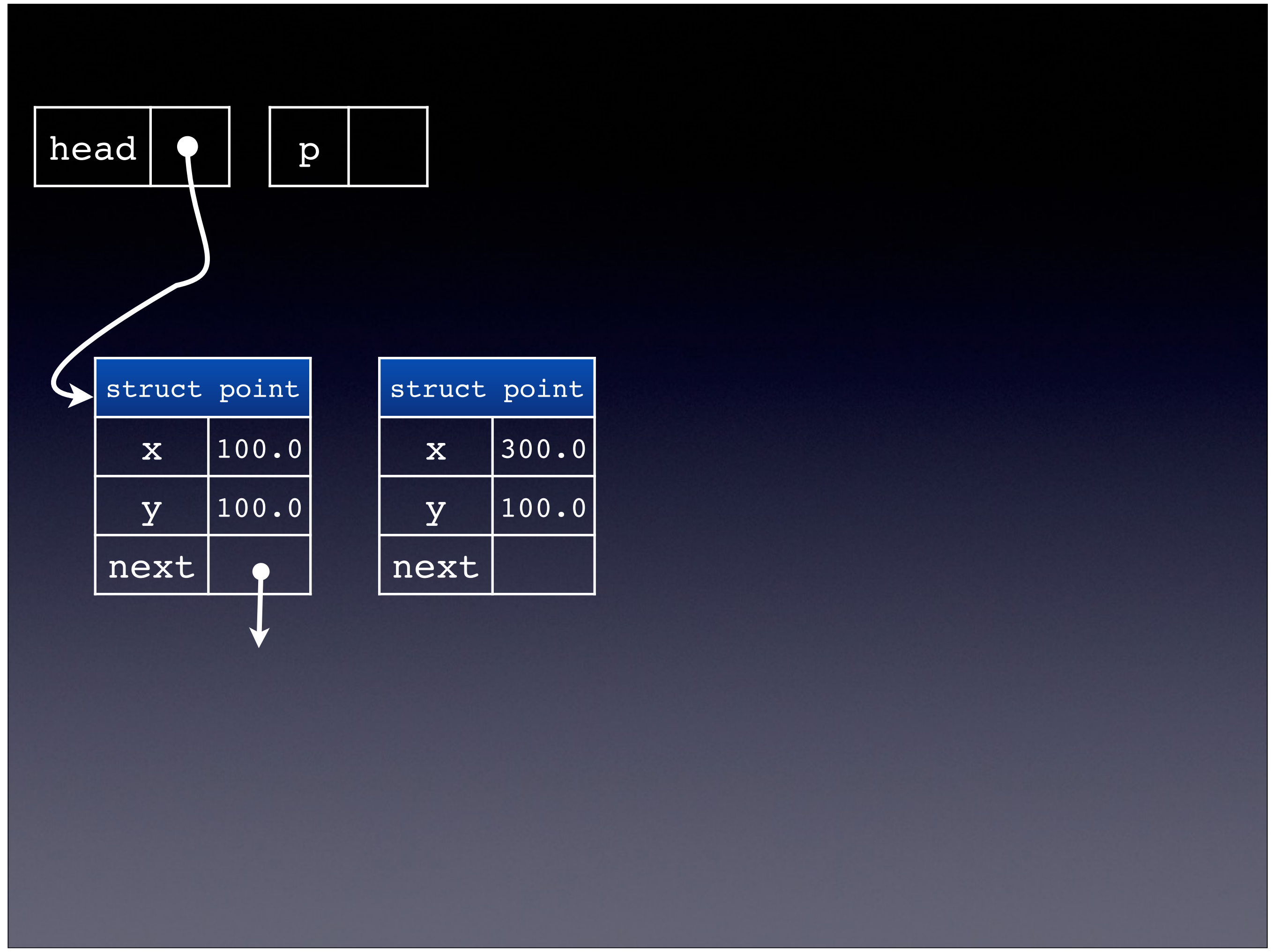
- If `head` is `NULL`, then previous algorithm will crash
- Will try to dereference `p` when it is `NULL`
- Need to treat this case differently
- Simple, just check whether head is `NULL`
- If so, set it to point to new `struct`

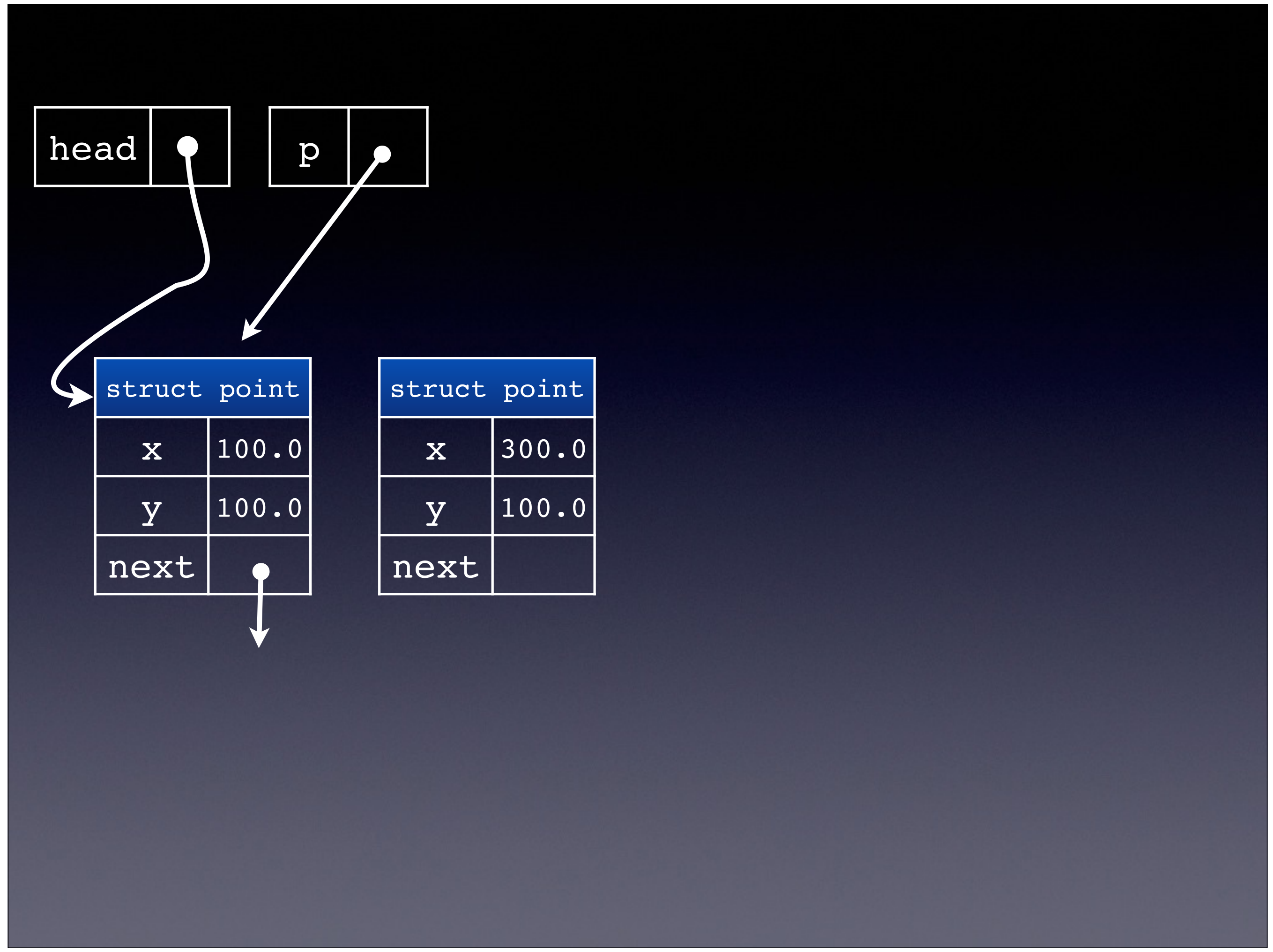

head

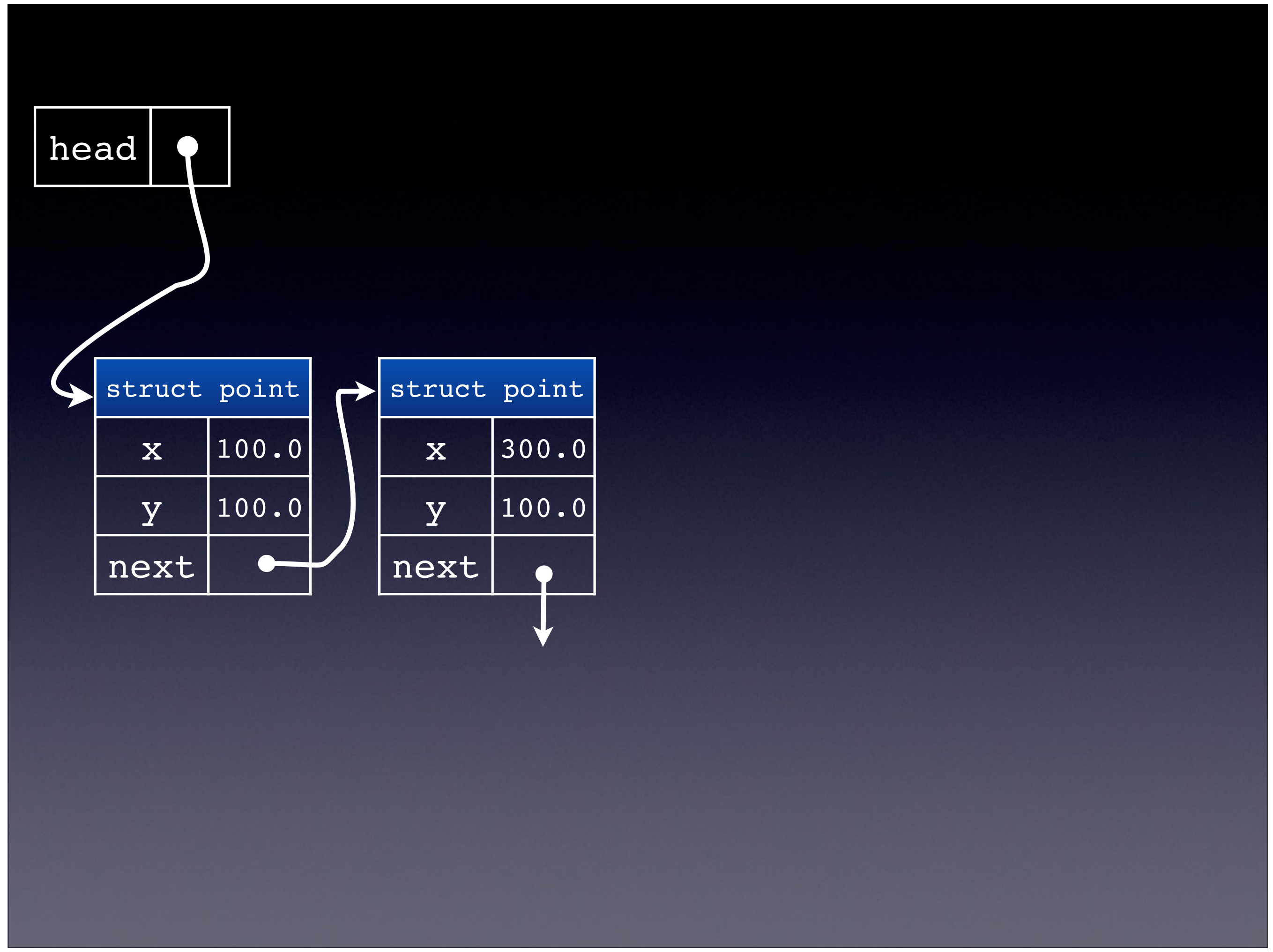

head
struct point
x
100.0
y
100.0
next

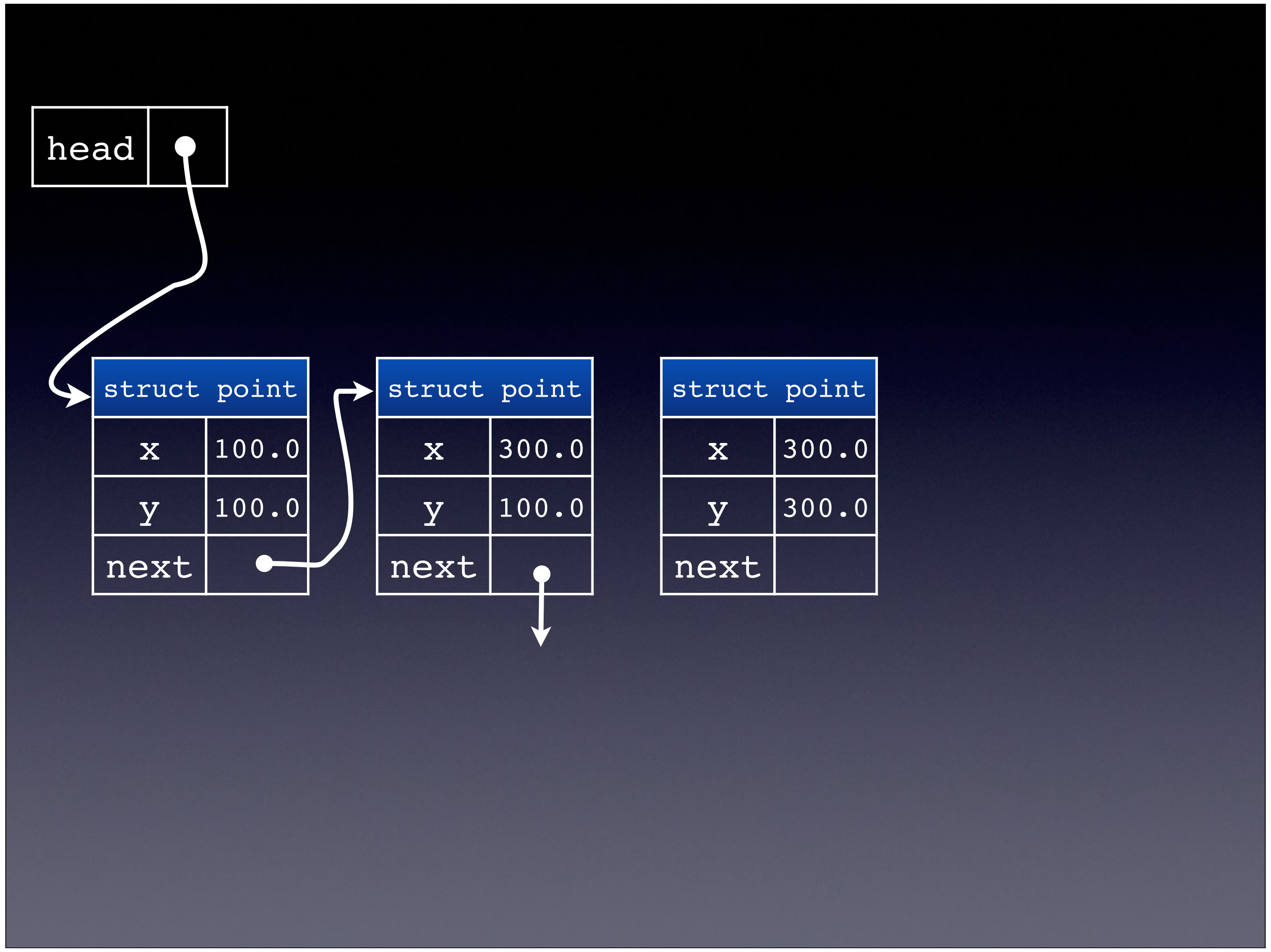

head
struct point
x
100.0
y
100.0
next

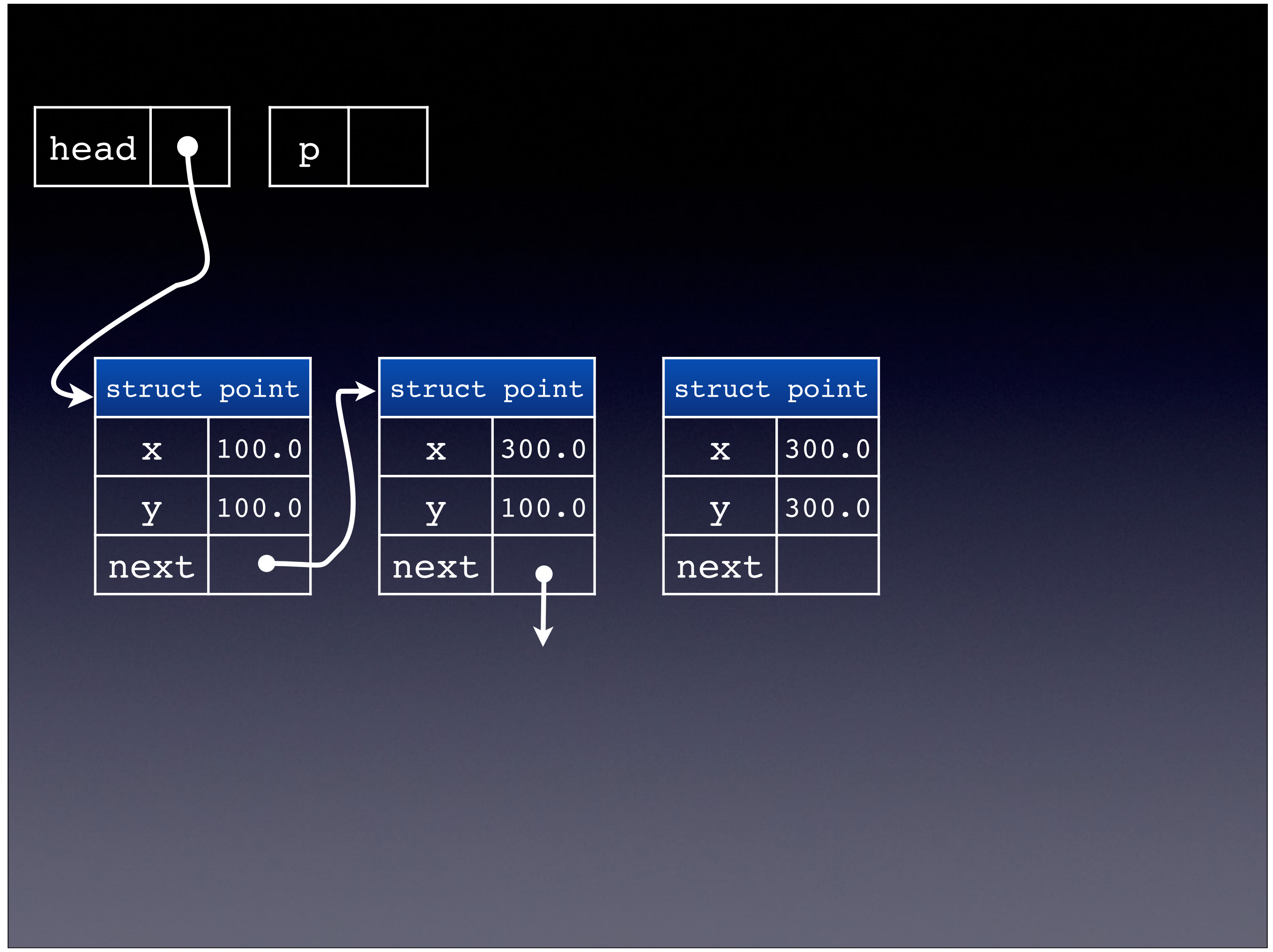

head
struct point
x
100.0
y
100.0
next
struct point
x
300.0
y
100.0
next


head
p
struct point
x
100.0
y
100.0
next
struct point
x
300.0
y
100.0
next

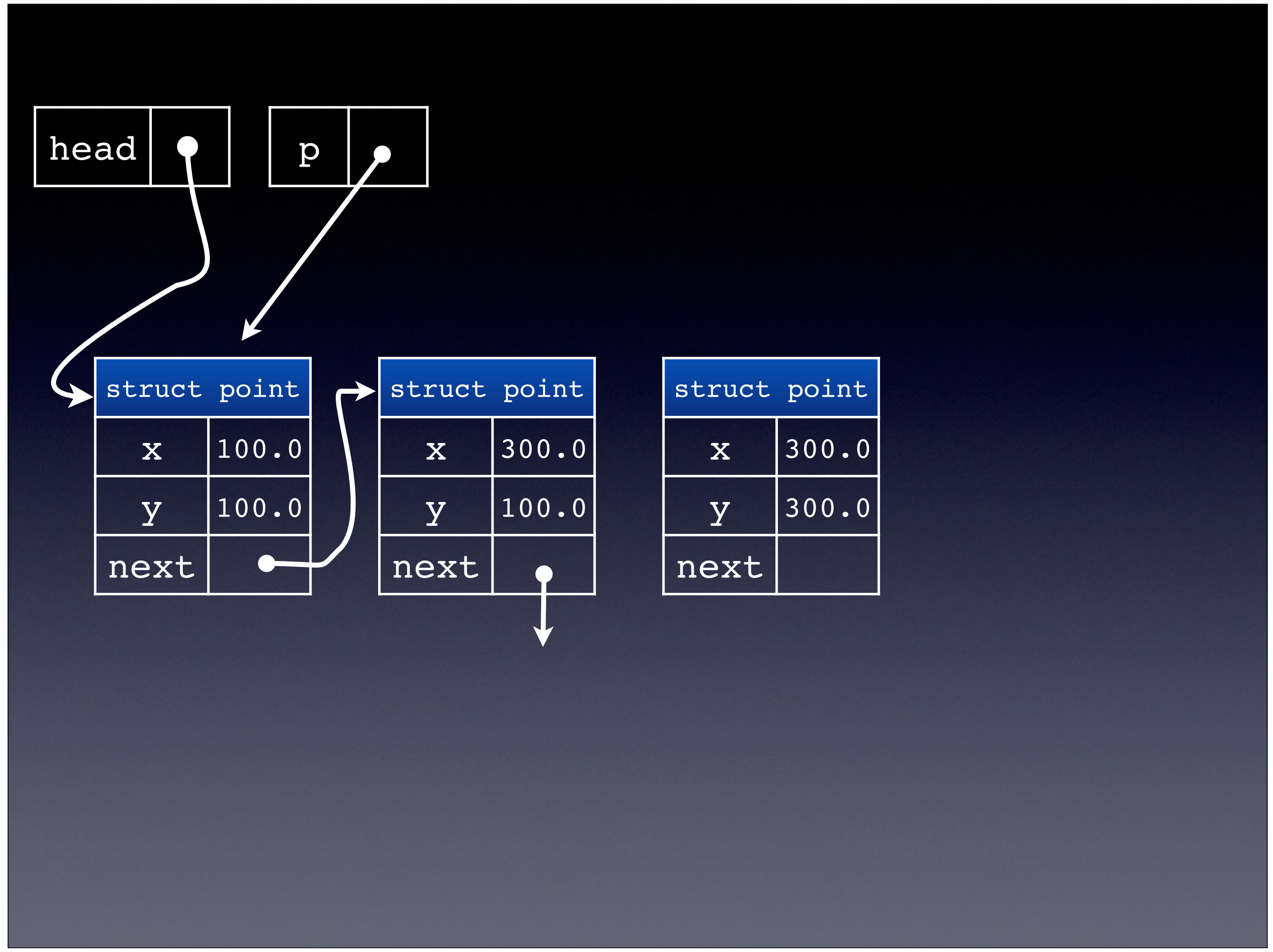

head
p
struct point
x
100.0
y
100.0
next
struct point
x
300.0
y
100.0
next

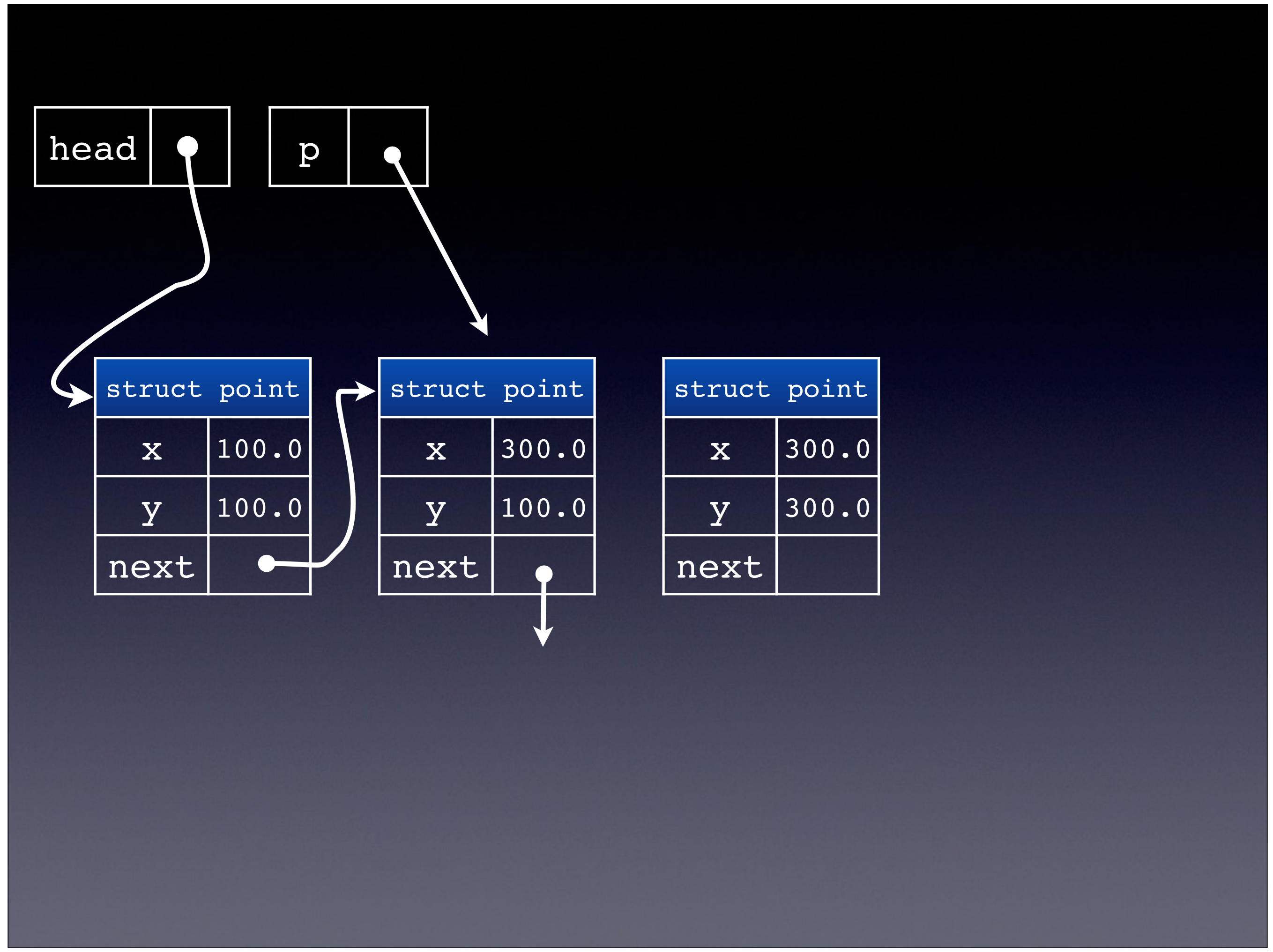

head
struct point
x
100.0
y
100.0
next
struct point
x
300.0
y
100.0
next

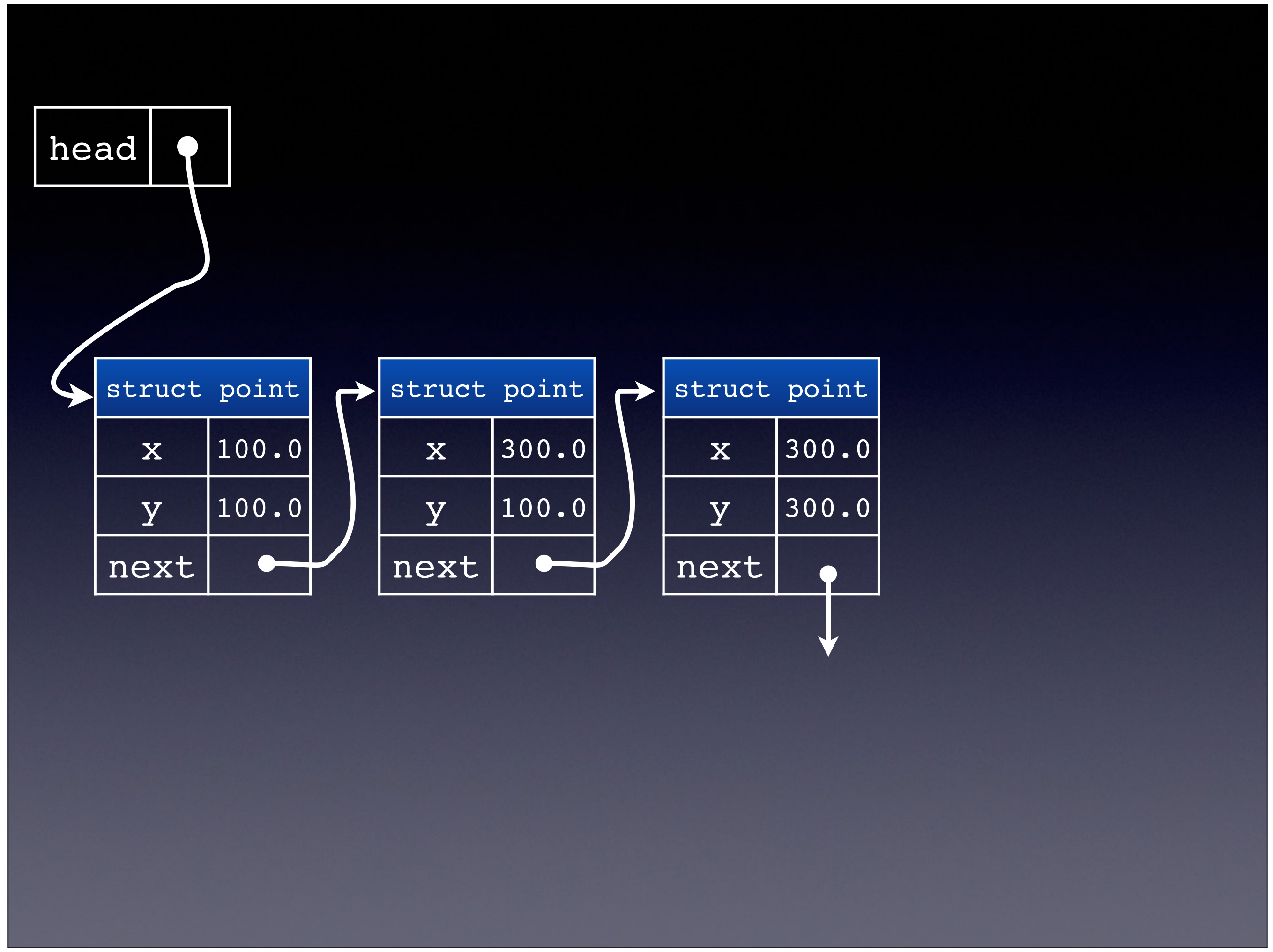

head
struct point
x
100.0
y
100.0
next
struct point
x
300.0
y
100.0
next
struct point
x
300.0
y
300.0
next

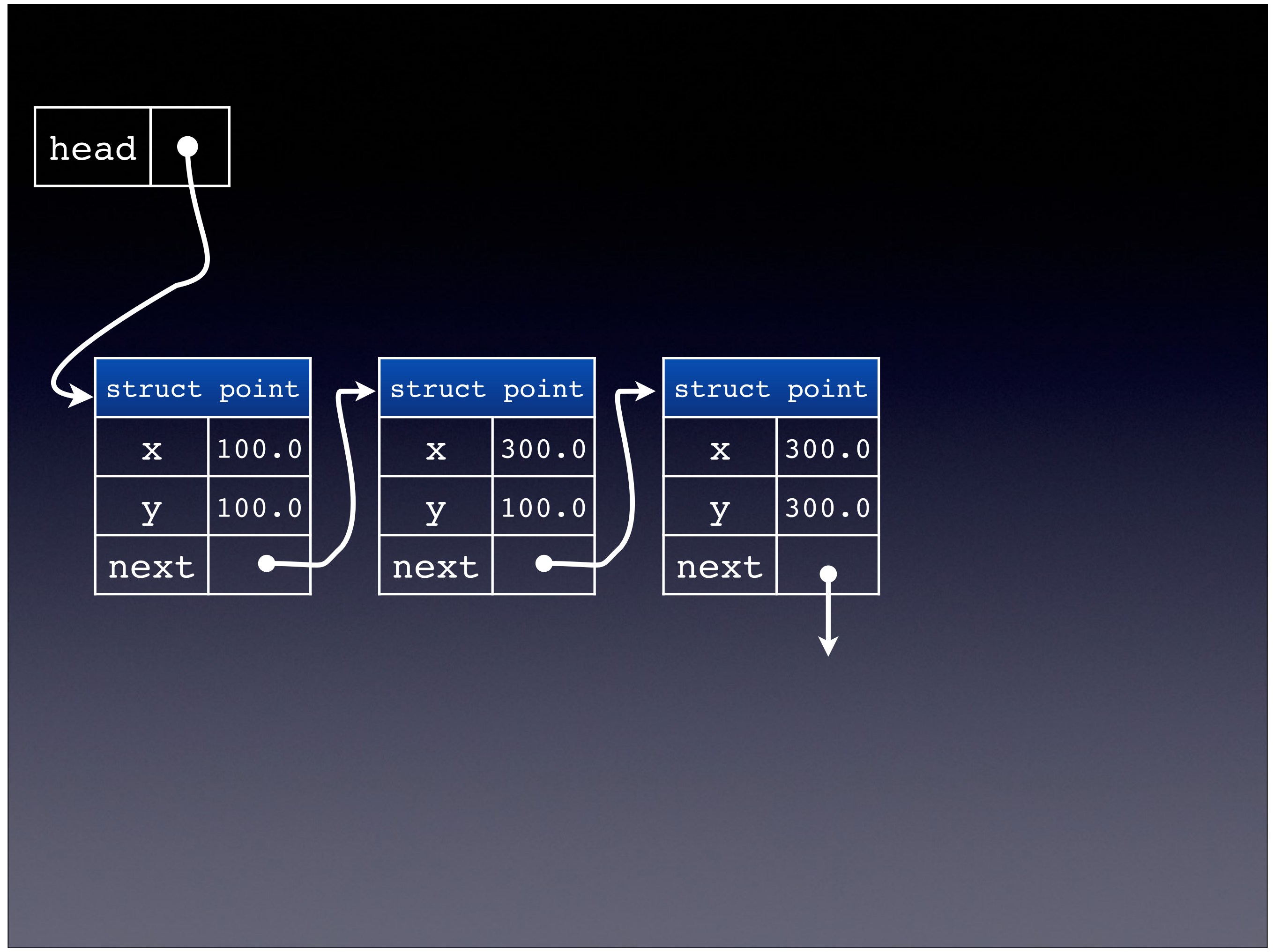


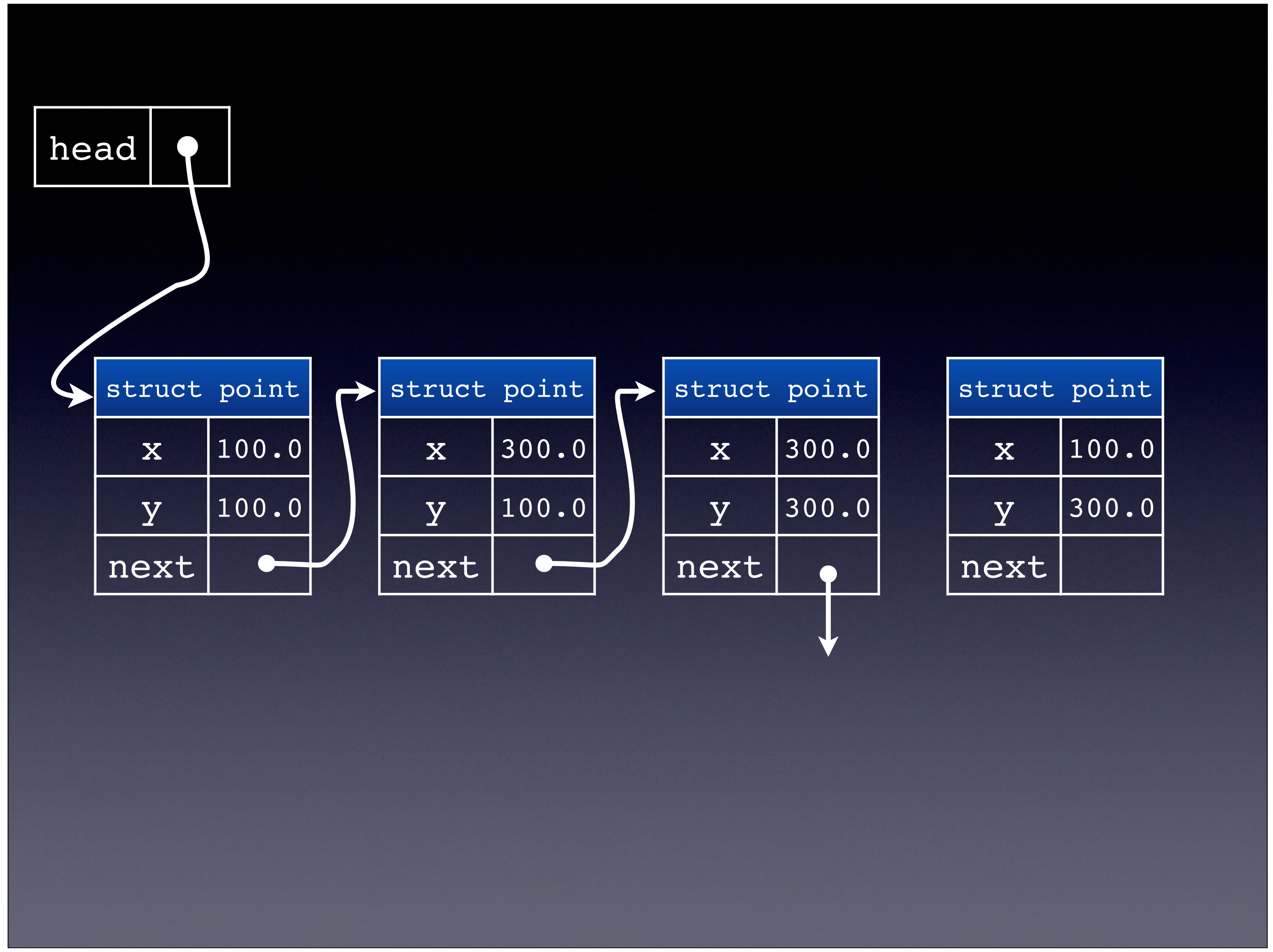
head
p
struct point
x
100.0
y
100.0
next
struct point
x
300.0
y
100.0
next
struct point
x
300.0
y
300.0
next

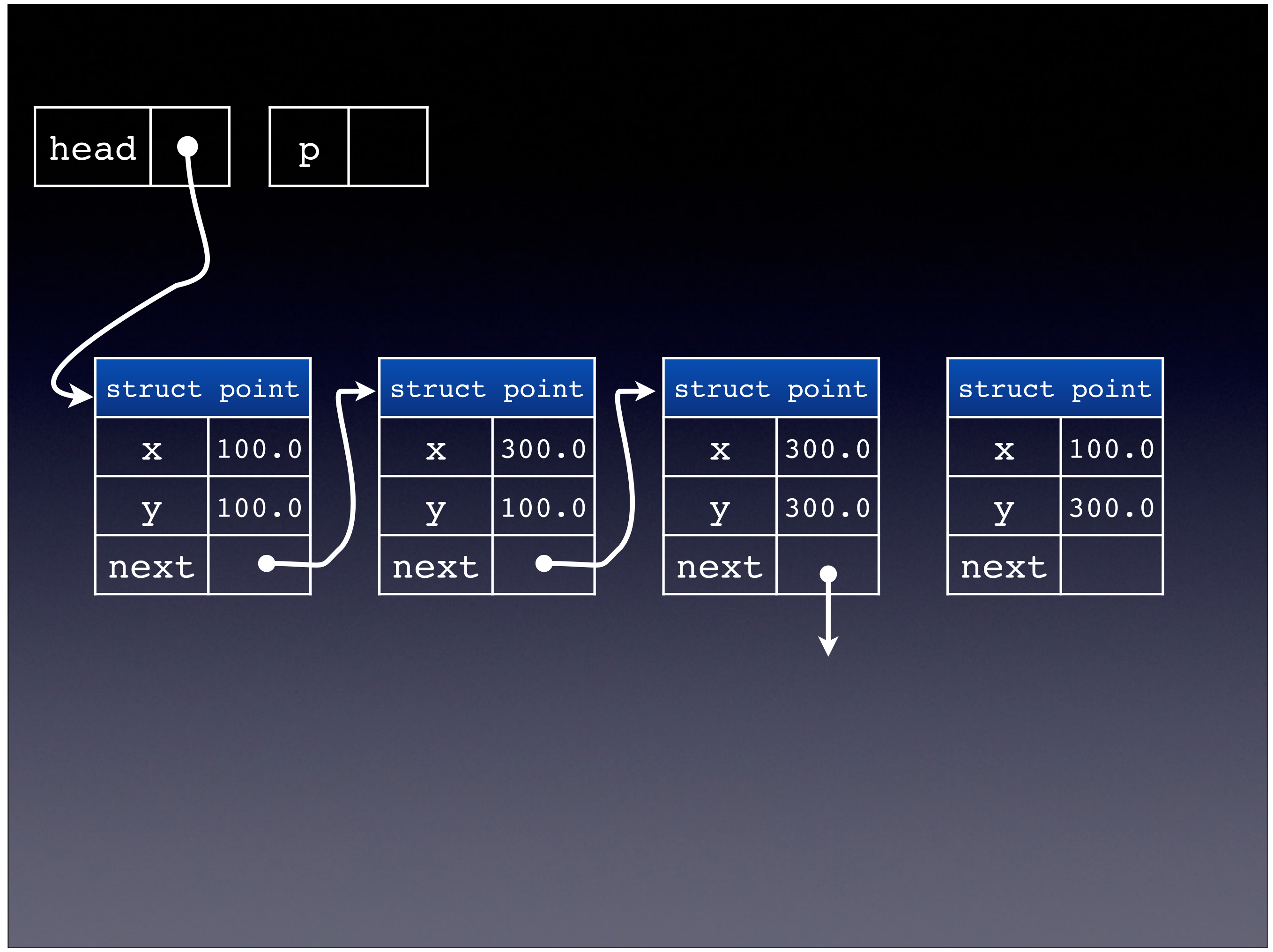

head
p
struct point
x
100.0
y
100.0
next
struct point
x
300.0
y
100.0
next
struct point
x
300.0
y
300.0
next

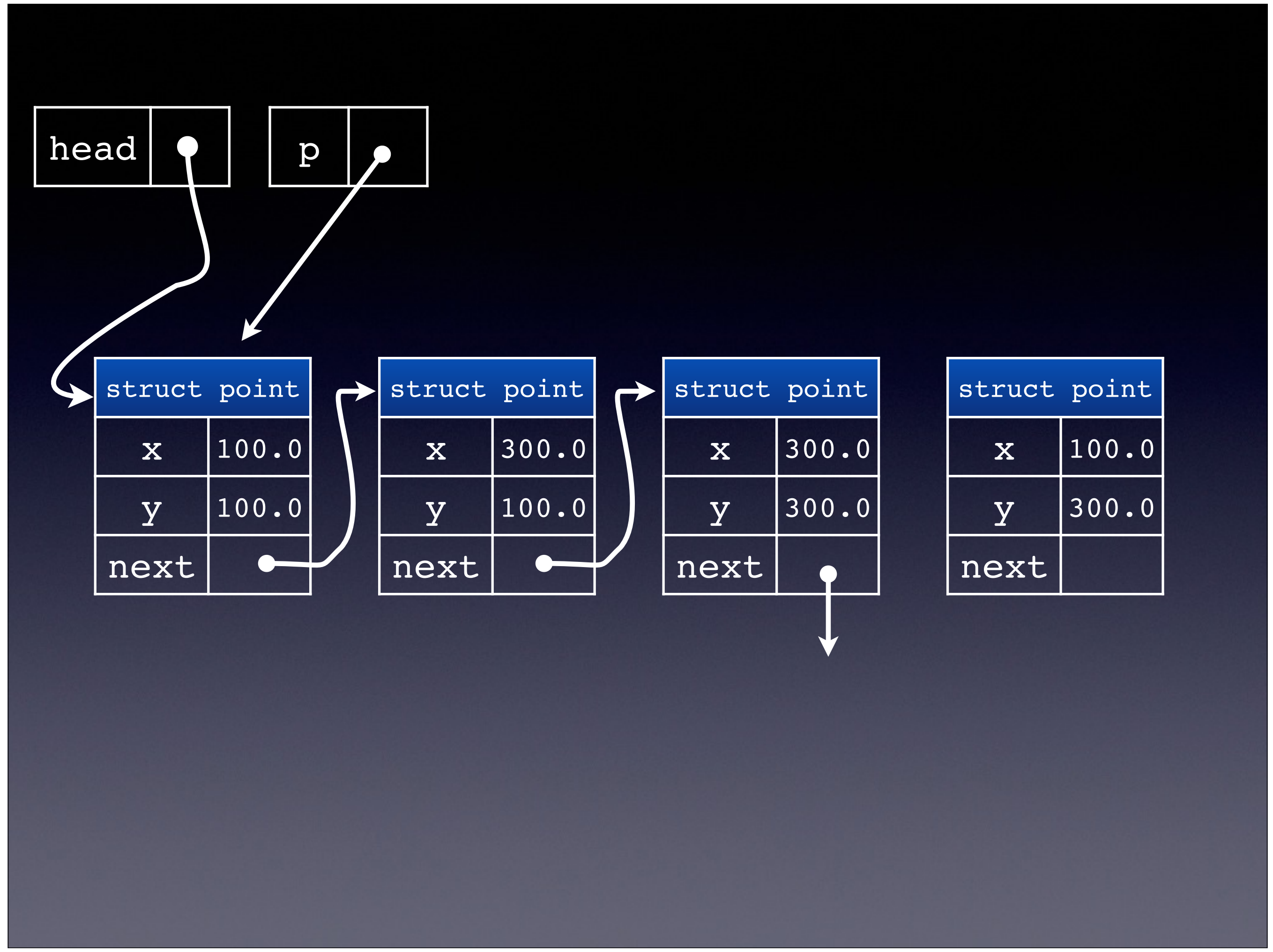

head
p
struct point
x
100.0
y
100.0
next
struct point
x
300.0
y
100.0
next
struct point
x
300.0
y
300.0
next

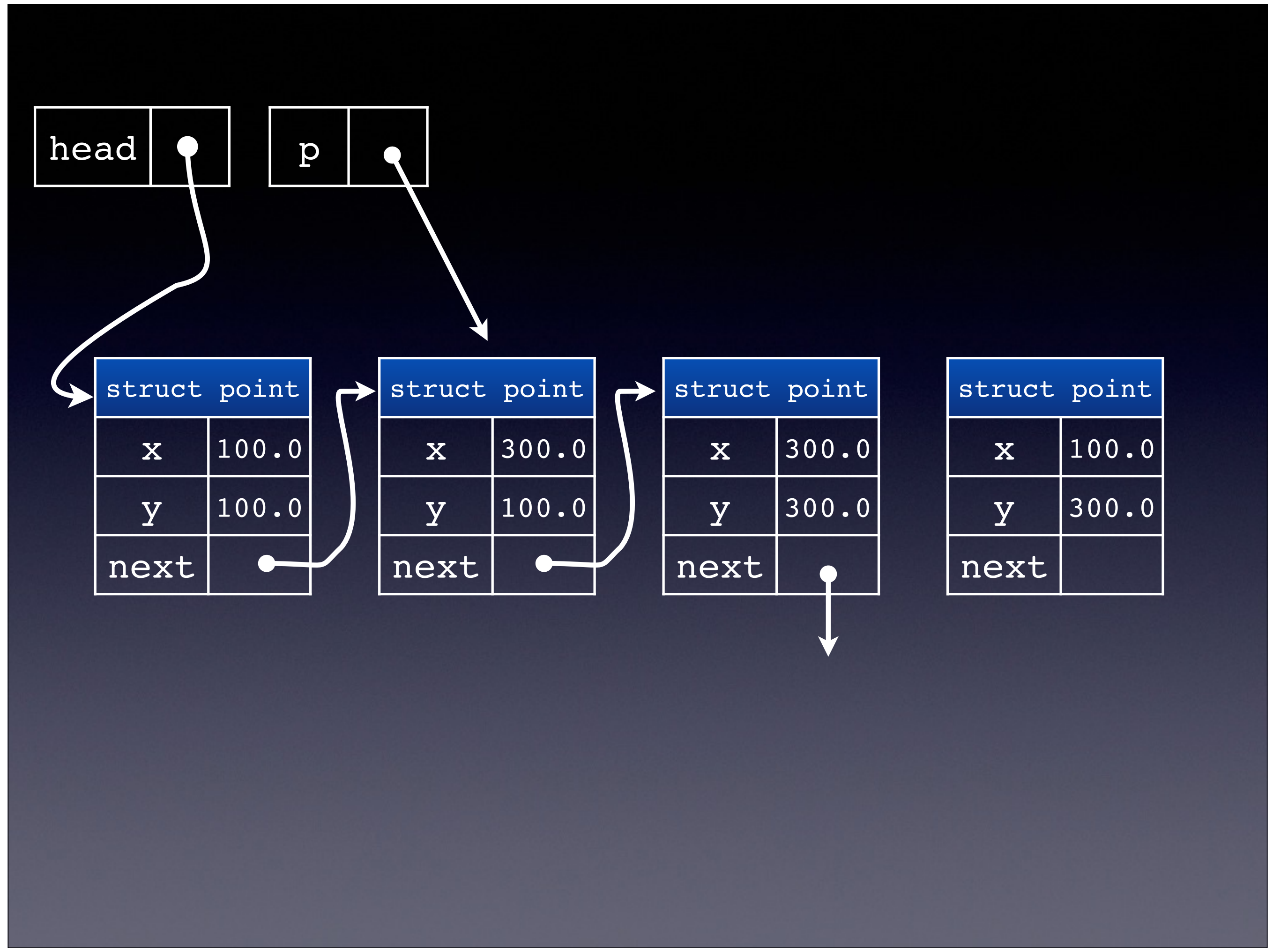

head
struct point
x
100.0
y
100.0
next
struct point
x
300.0
y
100.0
next
struct point
x
300.0
y
300.0
next

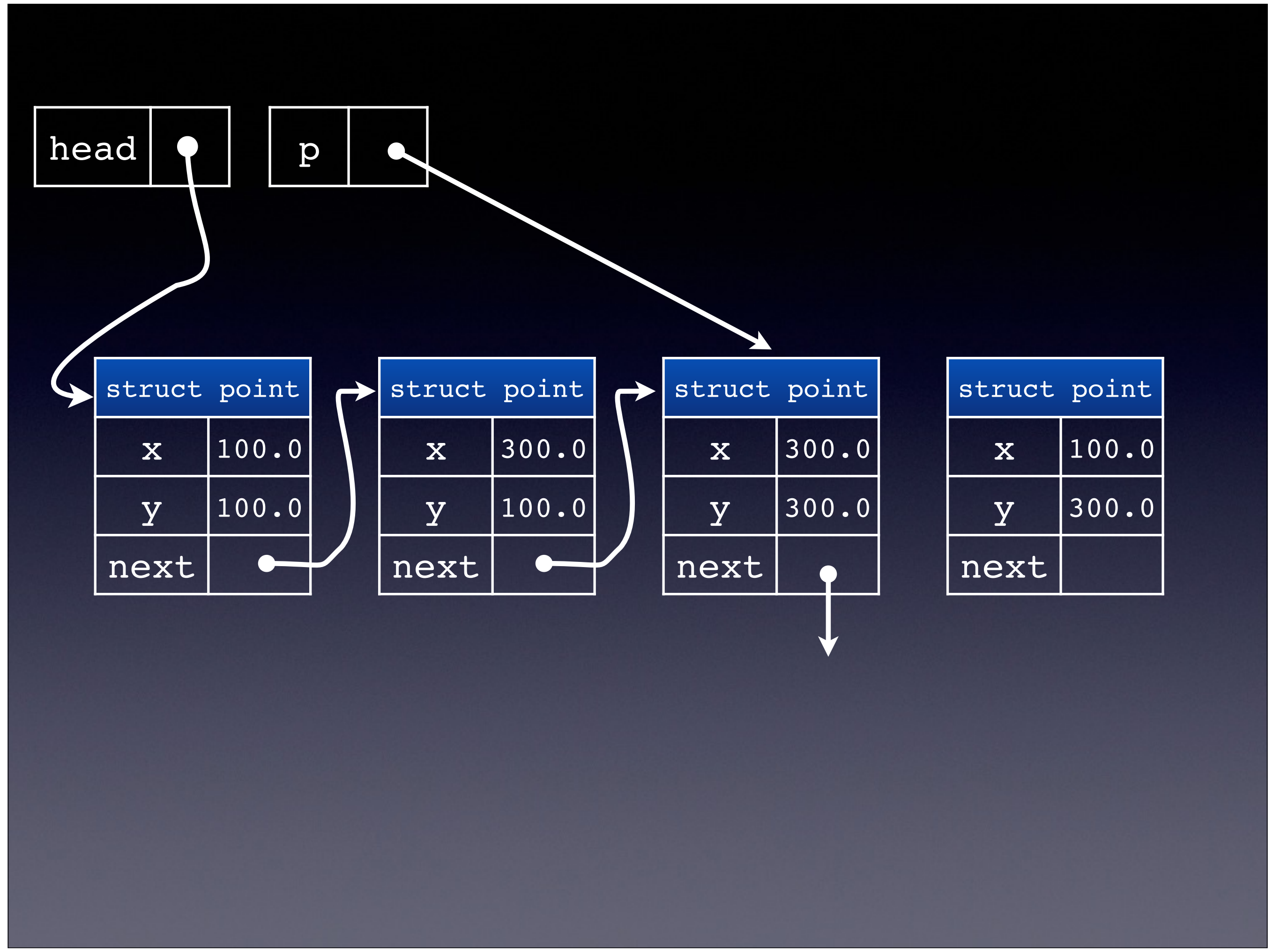

head
struct point
x
100.0
y
100.0
next
struct point
x
300.0
y
100.0
next
struct point
x
300.0
y
300.0
next

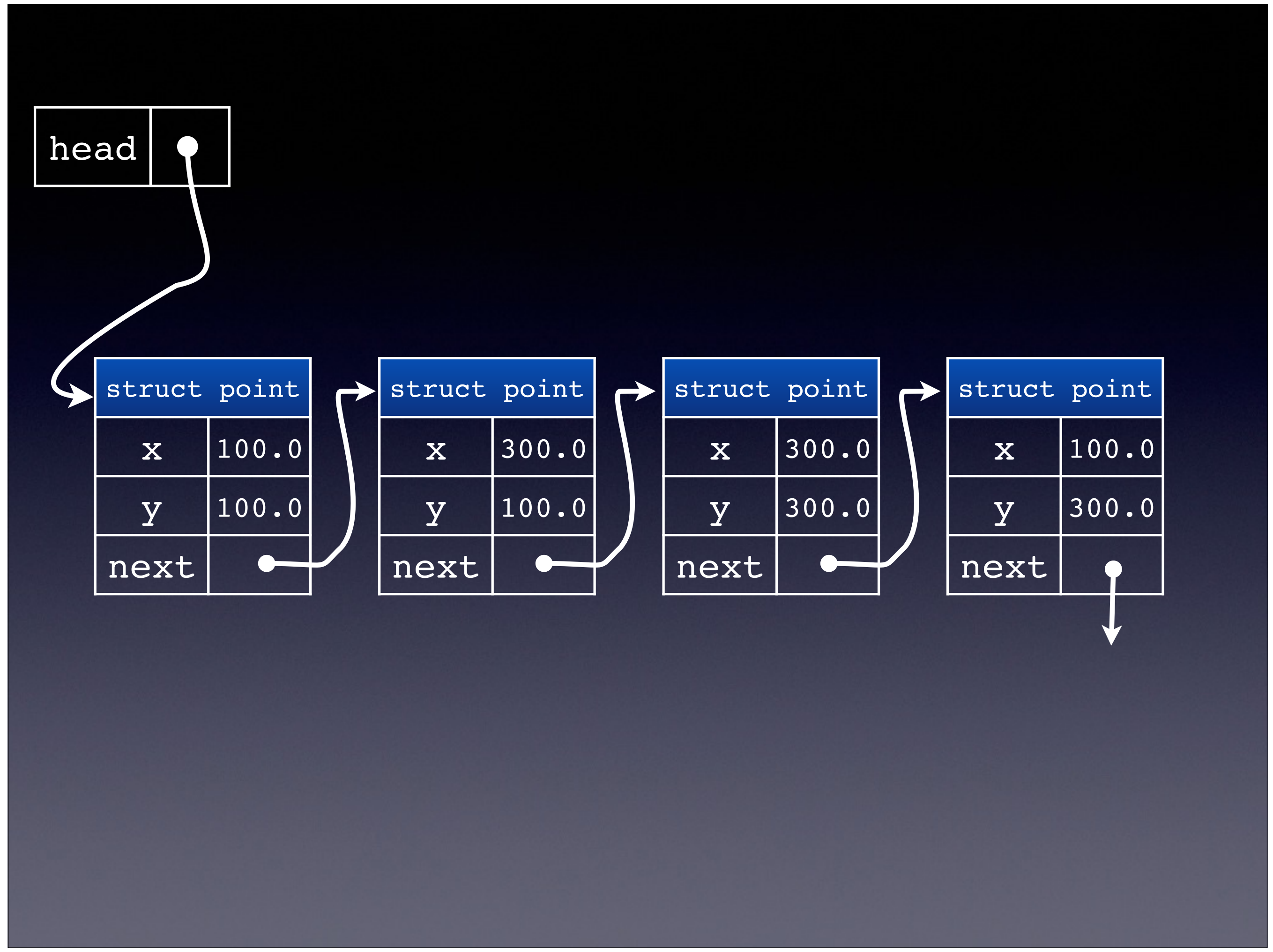

head
struct point
x
100.0
y
100.0
next
struct point
x
300.0
y
100.0
next
struct point
x
300.0
y
300.0
next
struct point
x
100.0
y
300.0
next


head
p
struct point
x
100.0
y
100.0
next
struct point
x
300.0
y
100.0
next
struct point
x
300.0
y
300.0
next
struct point
x
100.0
y
300.0
next


head
p
struct point
x
100.0
y
100.0
next
struct point
x
300.0
y
100.0
next
struct point
x
300.0
y
300.0
next
struct point
x
100.0
y
300.0
next


head
p
struct point
x
100.0
y
100.0
next
struct point
x
300.0
y
100.0
next
struct point
x
300.0
y
300.0
next
struct point
x
100.0
y
300.0
next


head
p
struct point
x
100.0
y
100.0
next
struct point
x
300.0
y
100.0
next
struct point
x
300.0
y
300.0
next
struct point
x
100.0
y
300.0
next



head
struct point
x
100.0
y
100.0
next
struct point
x
300.0
y
100.0
next
struct point
x
300.0
y
300.0
next
struct point
x
100.0
y
300.0
next

Adding at the end

- Quite slow to add things at the end of a linked list
- Each item slows down adding the next one since we have to visit one more item
- Can speed it up by keeping a pointer to the last item as well