The Linked List

Steven R. Bagley

Linked Lists

- Linked lists are another data structure
- Stores a collection of items in order
- Unlike arrays, the size is not fixed
- And it is easy to add and remove things
- Each item points to the next item in the list
- Last item points at nothing (`NULL`)

Linked Lists

- Unlike arrays, no direct support in C
- But very easy to create one
- Use a normal `struct` with variables for data
- Also has an extra variable, a pointer to the next `struct`, e.g.
  `struct point *next;`
- Almost always called `next`

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

Note it must be a pointer otherwise you'd create an infinitely big struct

The beginning

- Need a way to find the start of the list
- Use a normal pointer to a `struct` for this
- This pointer can also be `NULL`
- This is used to signify an empty list
- Sometimes this variable is called `head` (or similar variation)

The rest

- Each `struct` contains a pointer to the next link in the chain
- Can be `NULL` to signify the end of the list
- Finding an item means traversing each node in the list before it
- Starting with the first node…

i.e. the one where we know where it is…

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

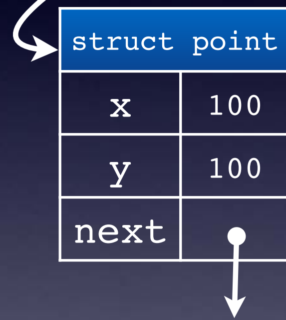
- Because we test whether `p->next` is `NULL` and not `p` itself
- When we reach the termination case, `p` will be pointing at the last item in the list
- Can then set `p->next` to point to our newly allocated `struct`

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
y
100
next

struct point	
x	100
y	100
next	

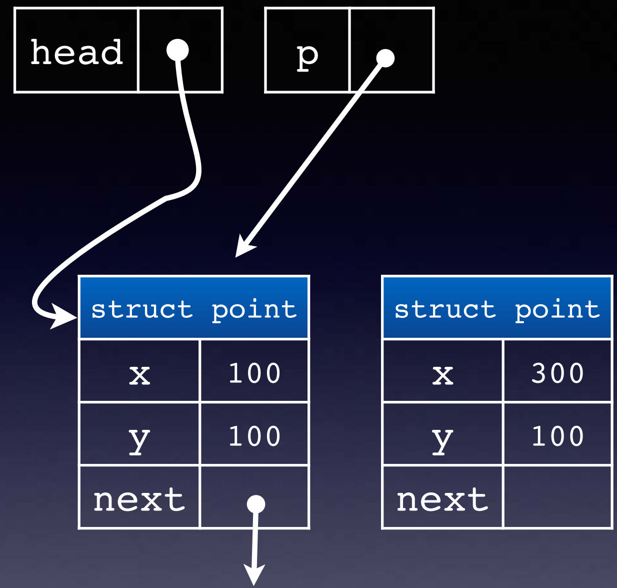
head

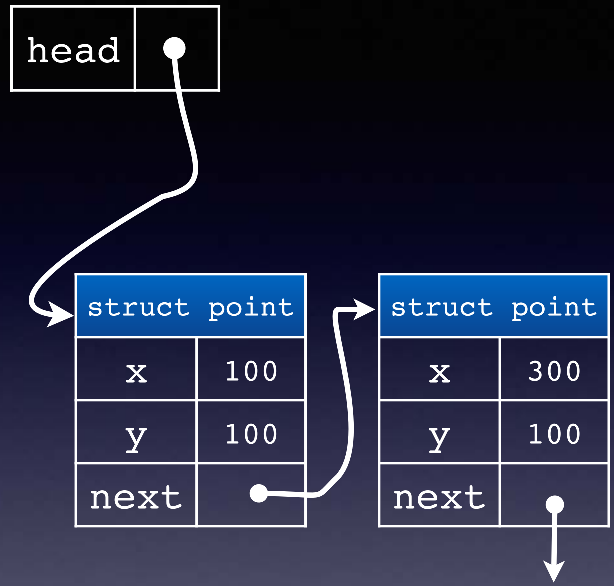
struct point	
x	100
y	100
next	

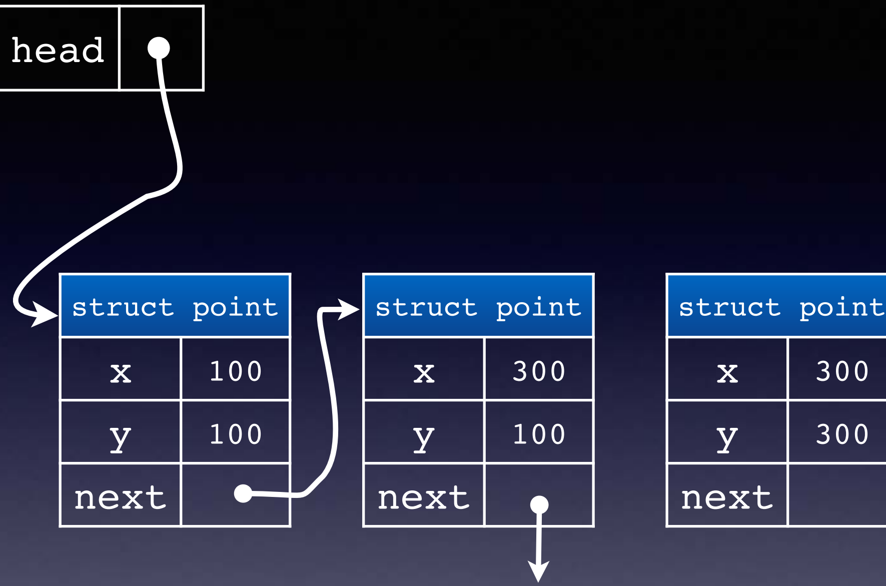
struct point	
x	300
y	100
next	

struct point	
x	100
y	100
next	

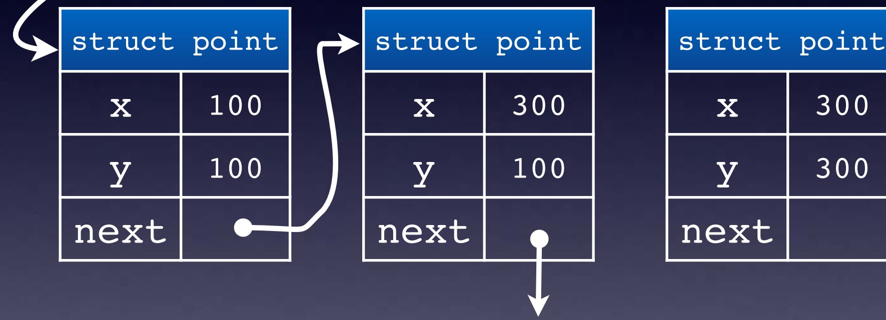
struct point	
x	300
y	100
next	

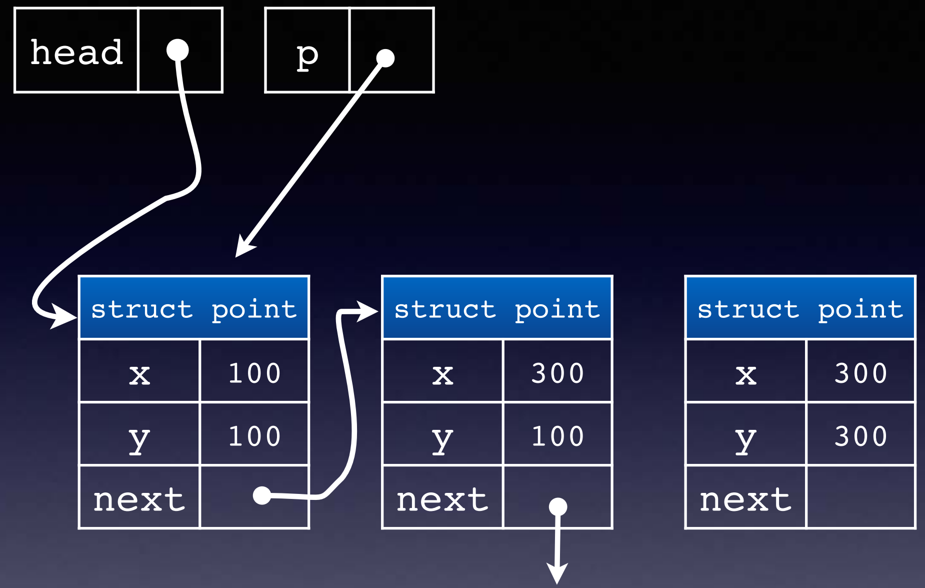

head
p
struct point
x
100
y
100
next
struct point
x
300
y
100
next

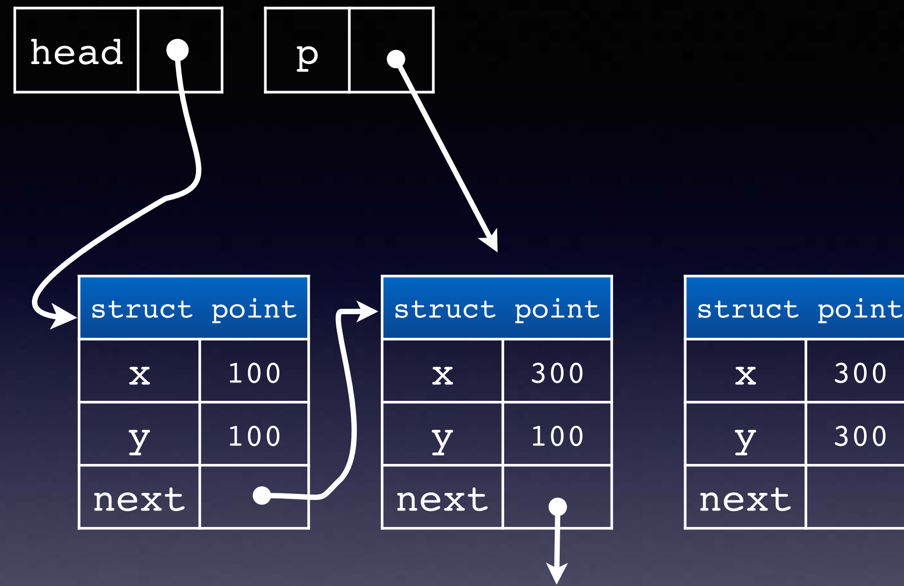

head
struct point
x
100
y
100
next
struct point
x
300
y
100
next

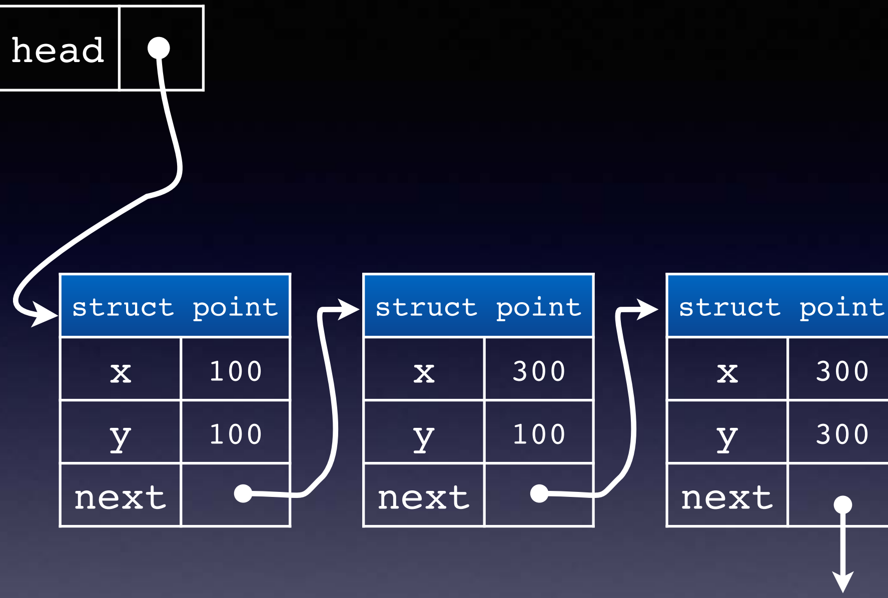

head
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next

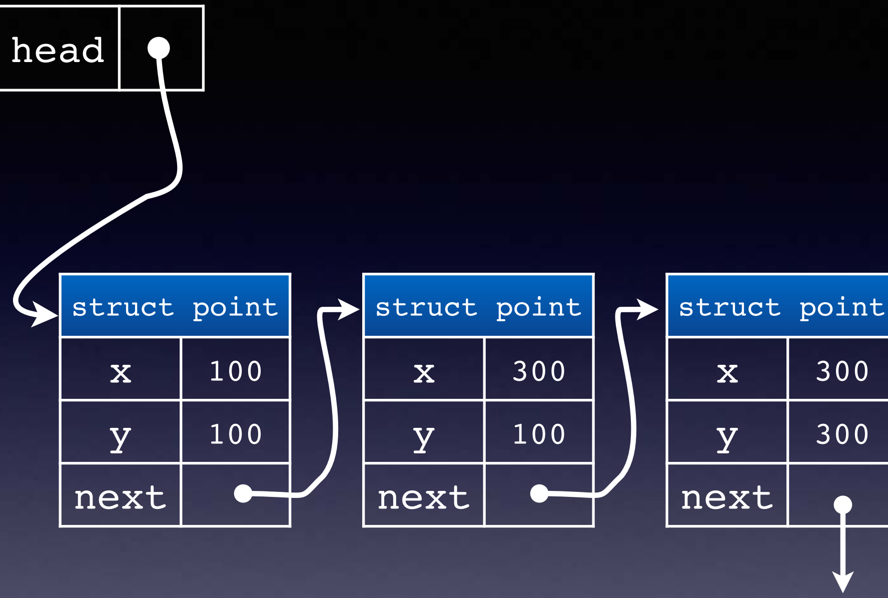
head
p

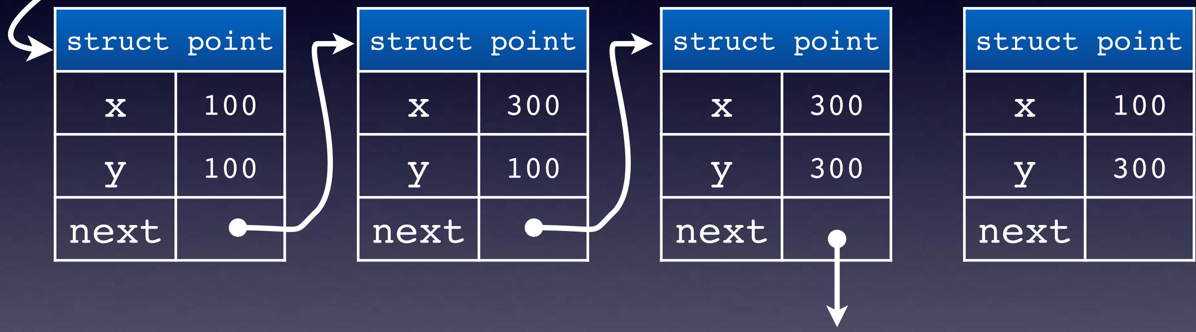

struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next


head
p
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next

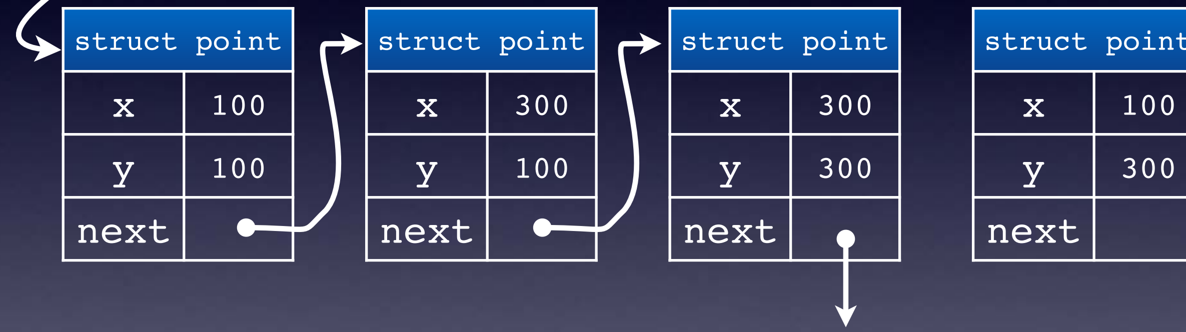

head
p
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next

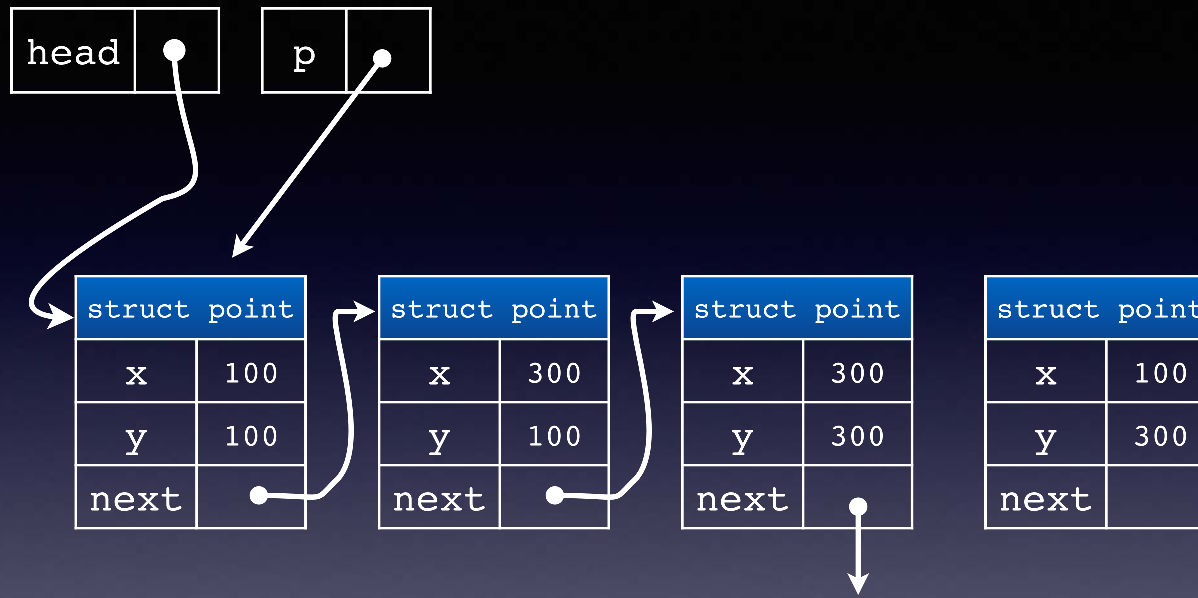

head
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next

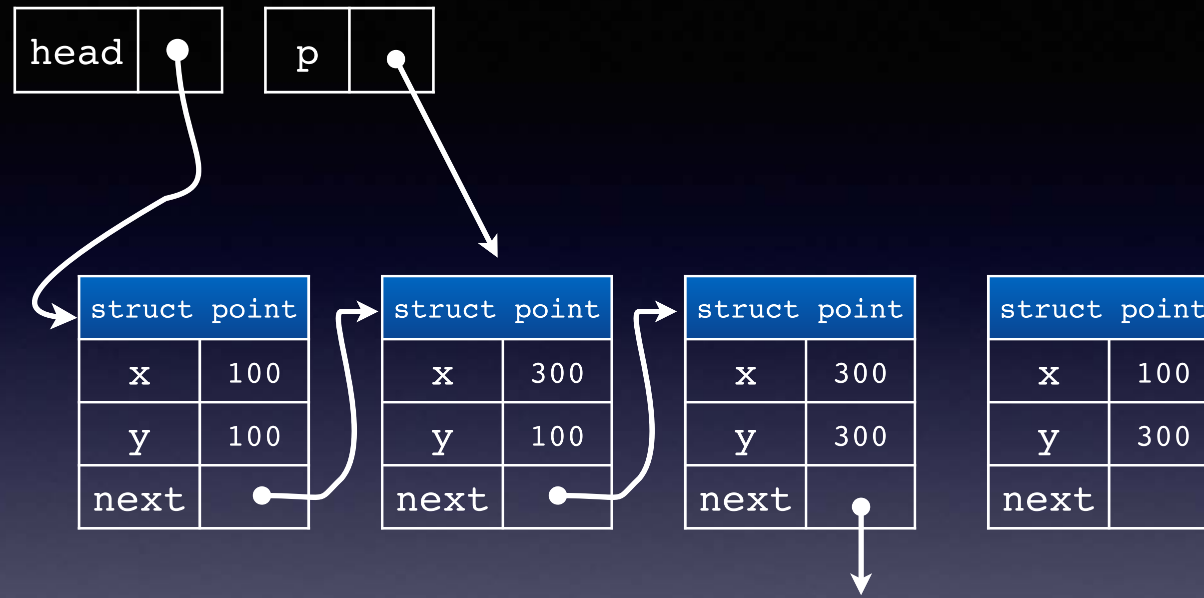

head
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next

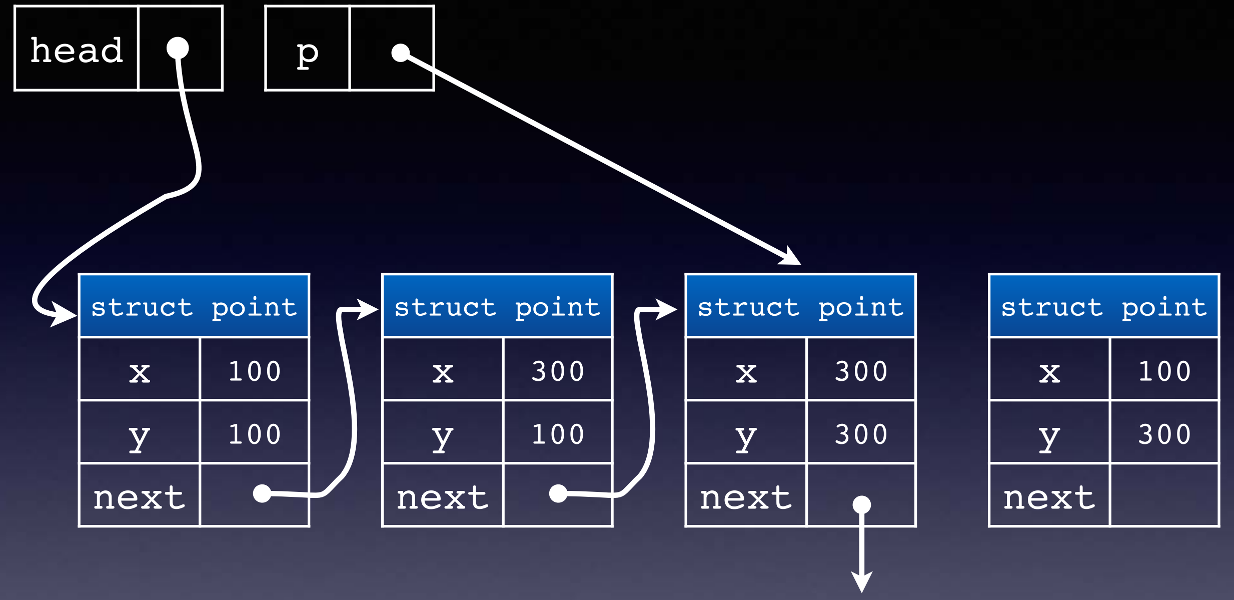

head
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next
struct point
x
100
y
300
next

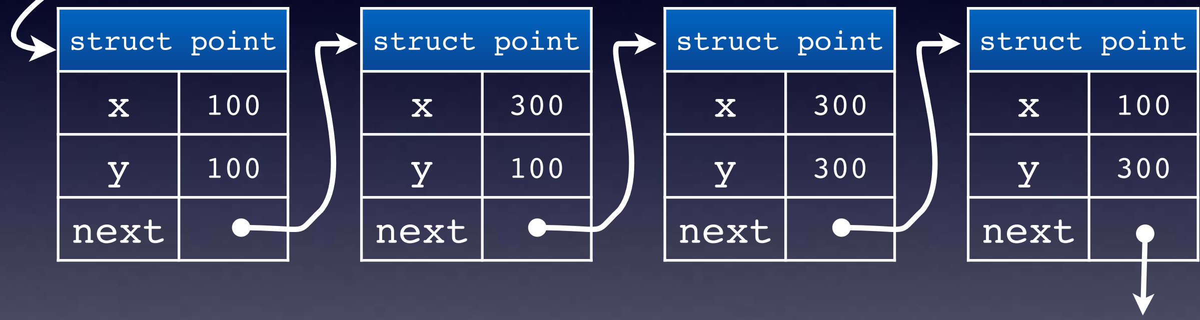
head
p


struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next
struct point
x
100
y
300
next


head
p
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next
struct point
x
100
y
300
next


head
p
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next
struct point
x
100
y
300
next


head
p
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next
struct point
x
100
y
300
next


head
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next
struct point
x
100
y
300
next

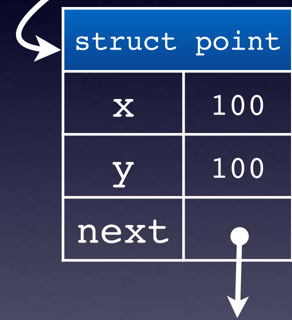
Adding at the end

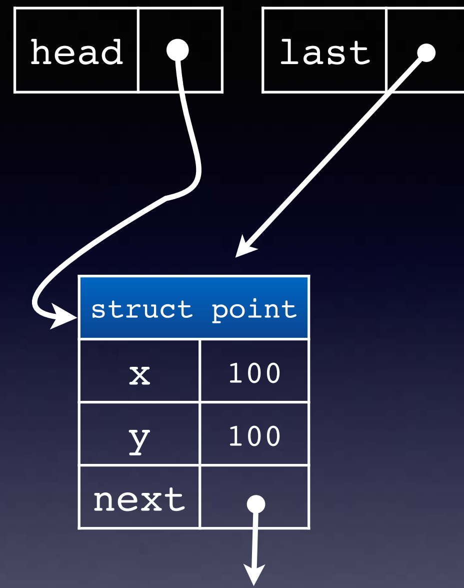
- Quite slow to add things at the end of a linked list
- Each item slows down adding the next one since we have to visit one more item
- Can speed it up by keeping a pointer to the last item as well

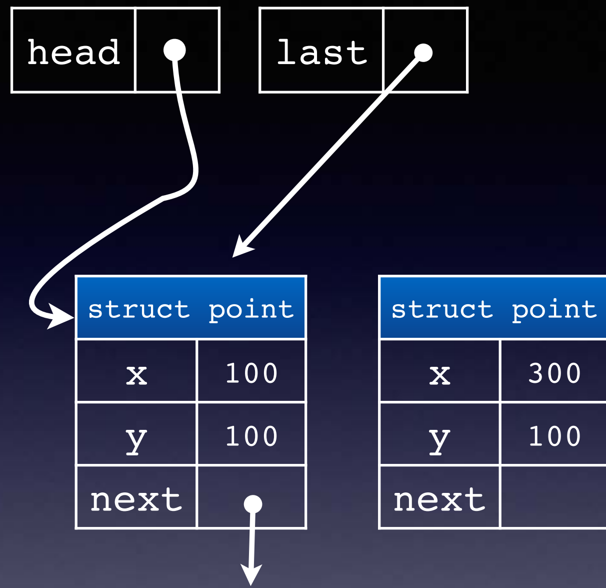
head

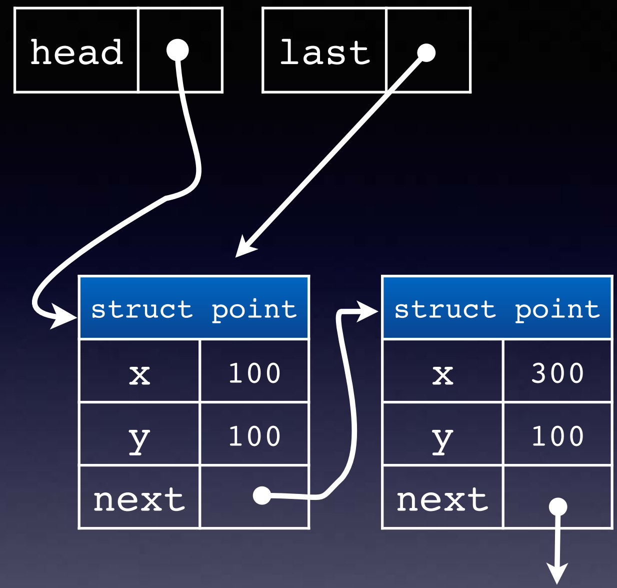
last

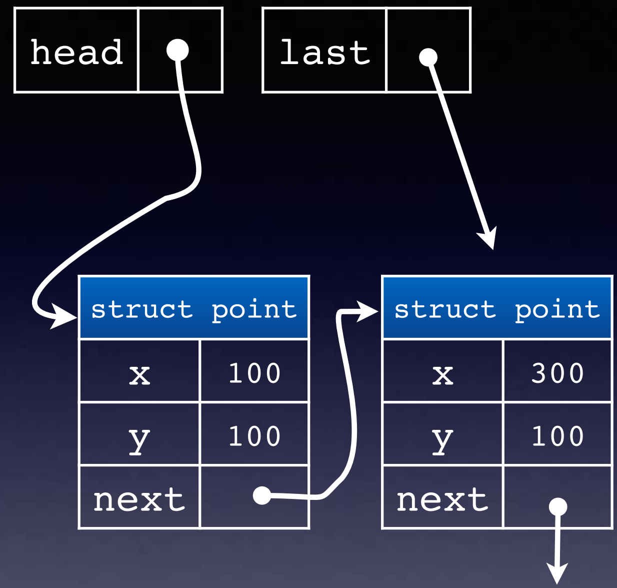
head
last

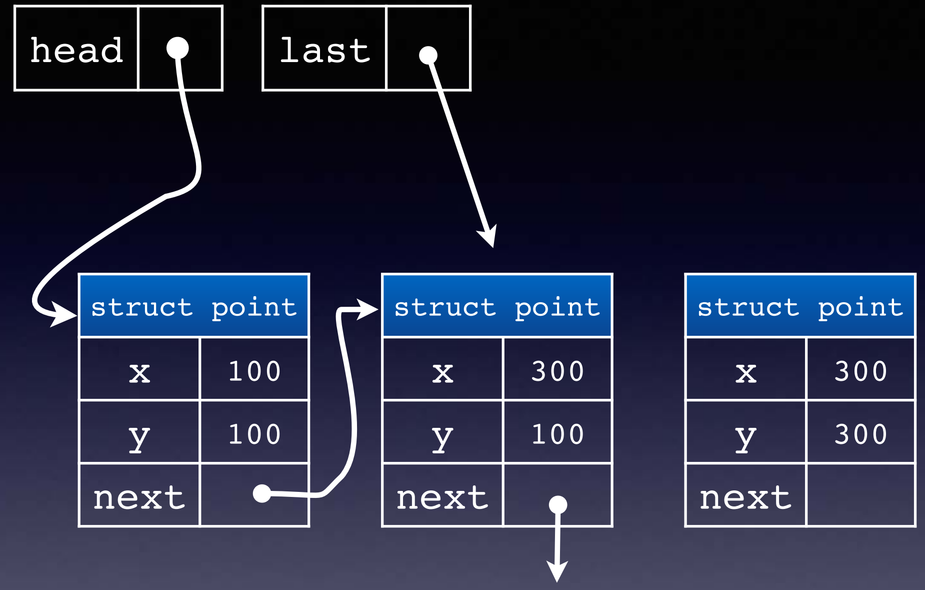

struct point
x
100
y
100
next

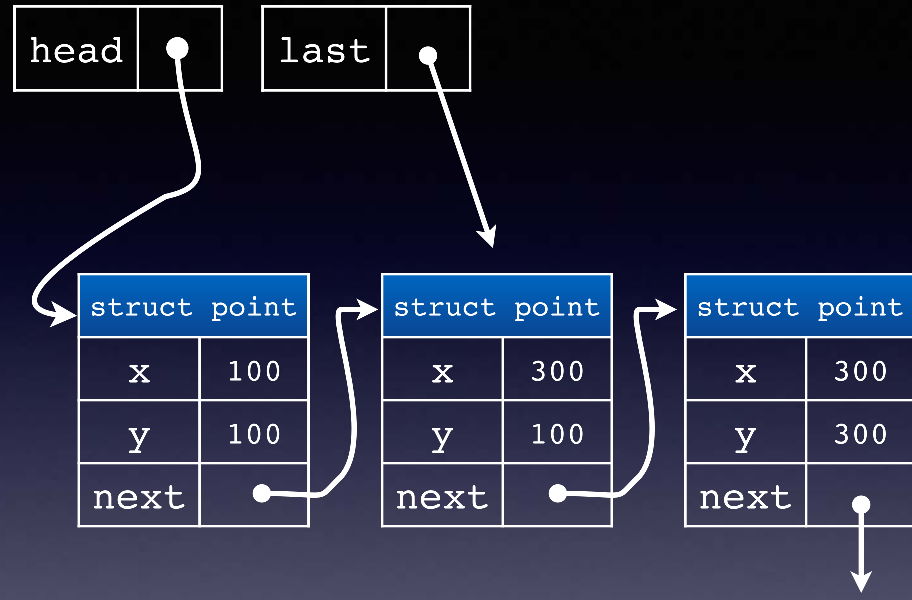

head
last
struct point
x
100
y
100
next

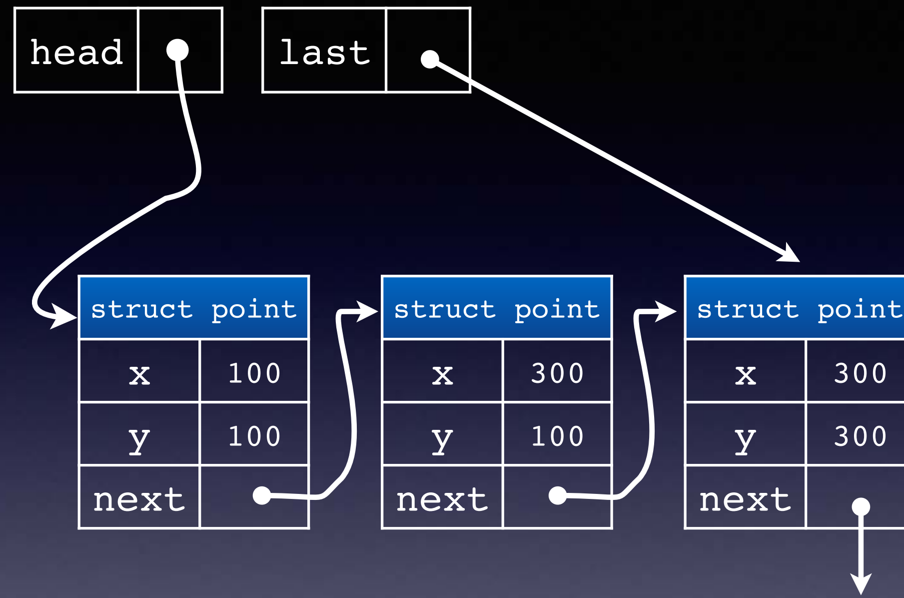

head
last
struct point
x
100
y
100
next
struct point
x
300
y
100
next

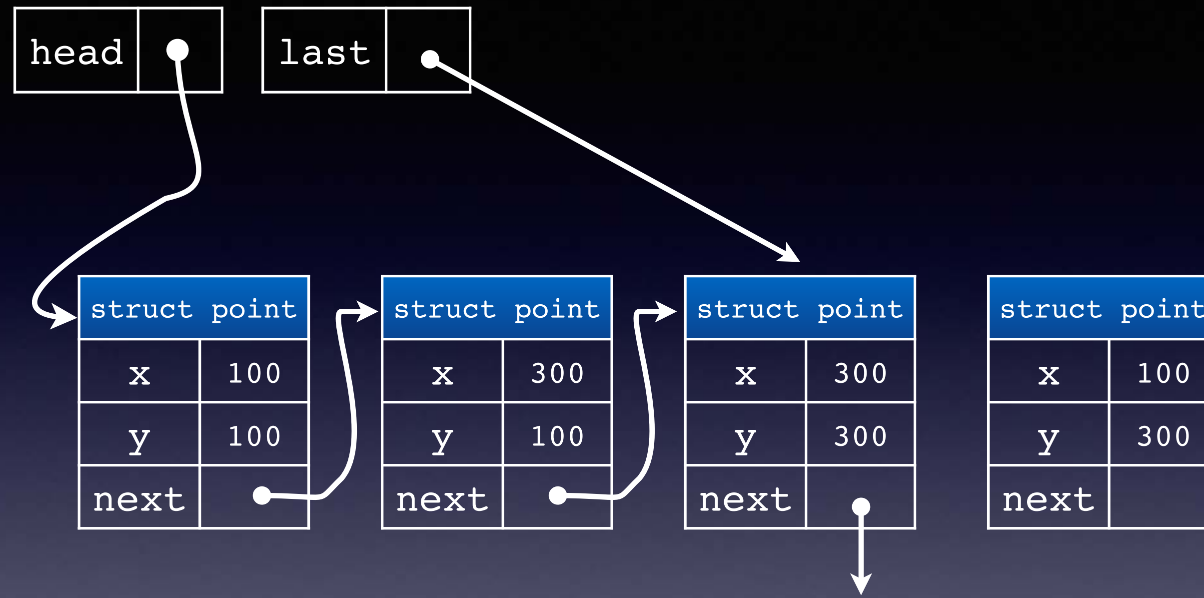

head
last
struct point
x
100
y
100
next
struct point
x
300
y
100
next

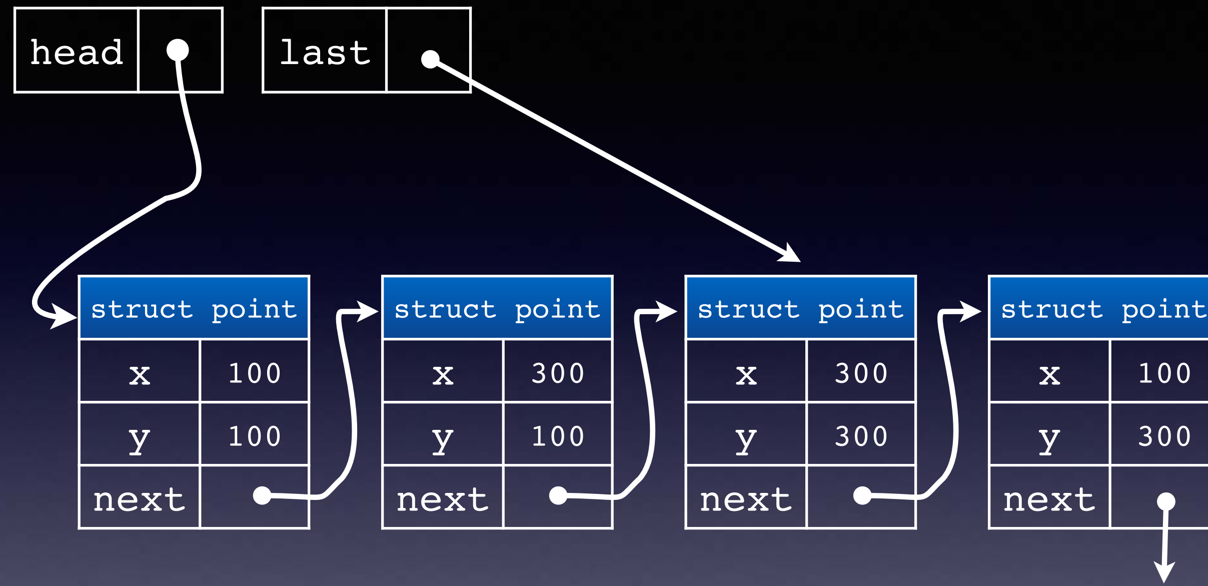

head
last
struct point
x
100
y
100
next
struct point
x
300
y
100
next

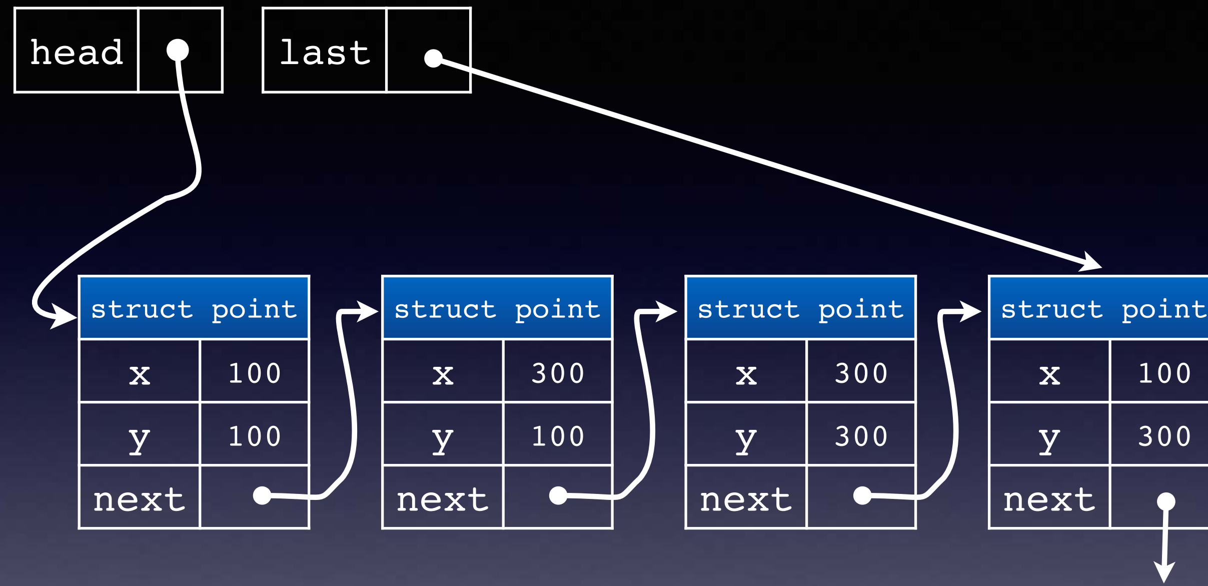

head
last
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next


head
last
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next


head
last
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next


head
last
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next
struct point
x
100
y
300
next


head
last
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next
struct point
x
100
y
300
next



head
last
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next
struct point
x
100
y
300
next

Adding at end

- `last` pointer means we never have to search for the end of the list
- So faster code, and more straightforward
- But another variable to keep track of…
- Use a function to add element to end…
- But how does it access `head`, and `last` variables?

Adding at the beginning

- Alternatively we can add things at the beginning
- Allocate new `struct`
- Set its `next` pointer to be the same as `head`
- Set `head` to point to the new `struct`
- But the order of items will be reversed

Walking the list

- Walking over the list and visiting each item is a very common process
- E.g. Counting the items in a list
- Find a specific thing (either by a value, or by position)
- Add things in the middle or at the end

Counting Items in a list

- Set `p` to equal `head`
- Set counter to zero
- While `p` is not `NULL`
    - Add one to counter
    - Set `p` to `p->next`

```
struct point *p = listHead;
int count = 0;

while(p != NULL)
{
    count++;

    p = p->next;
}
```

Not just limited to our struct point, can be used for any linked list

```
struct point *p = listHead;
int count = 0;

while(p != NULL)
{
    count++;

    p = p->next;
}
```

The code in white is common to all list walkers, the greyed out bits changed depending on the task.

Finding Items (positionally)

- Set `p` to equal `head` and set counter to zero
- While `p` is not `NULL`
  - If counter equals position, then `break`
  - Add one to counter
  - Set `p` to `p->next`
- If p is not `NULL`, then it points to the item

To search for a specific value test the data for that value rather than looking for a condition

```
int positionToFind = 4;

struct point *p = listHead;
int count = 0;

while(p != NULL)
{
    if(count == positionToFind)
         break;

    count++;
    p = p->next;
}
```

Not just limited to our struct point, can be used for any linked list

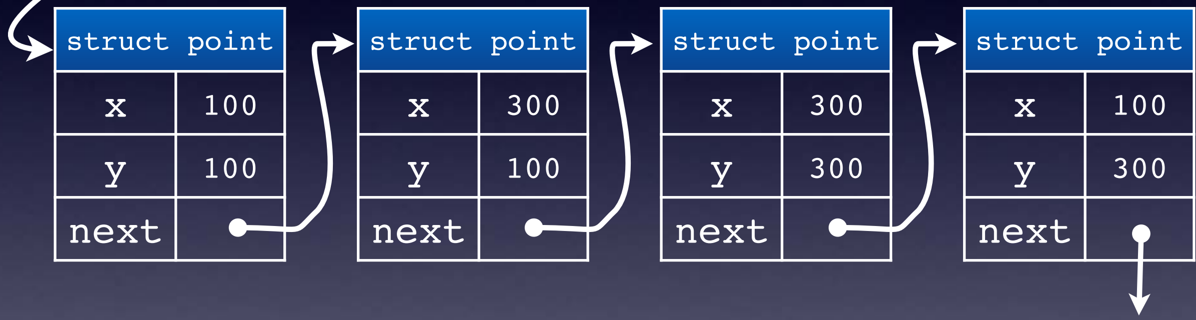
```
int positionToFind = 4;

struct point *p = listHead;
int count = 0;

while(p != NULL && count < positionToFind)
{
    count++;
    p = p->next;
}
```

Note I’ve used less than < and != -- doesn’t make much difference but slightly more defensive

Add in the middle

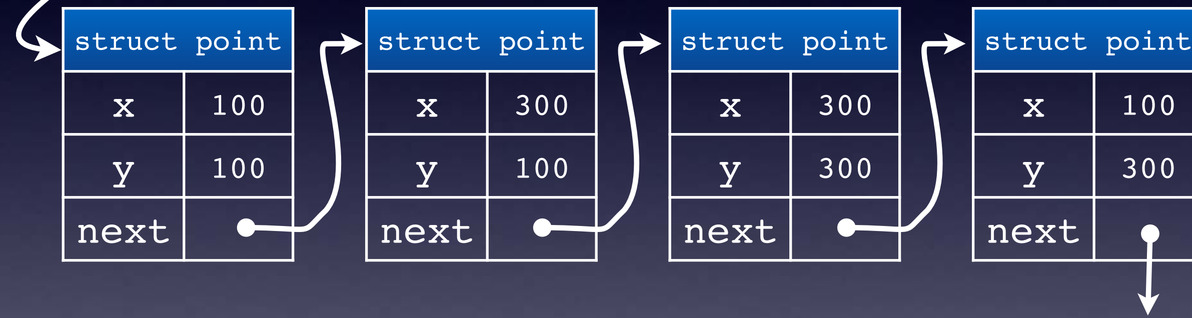
- Find the item before the position we want to add at (so if we want to add at position 2, find item at position 1) this is now in `p`
- Create new struct
- Set new `struct`'s next to point to `p->next`
- Set `p->next` to point to new `struct`
- New `struct` is now linked into the list

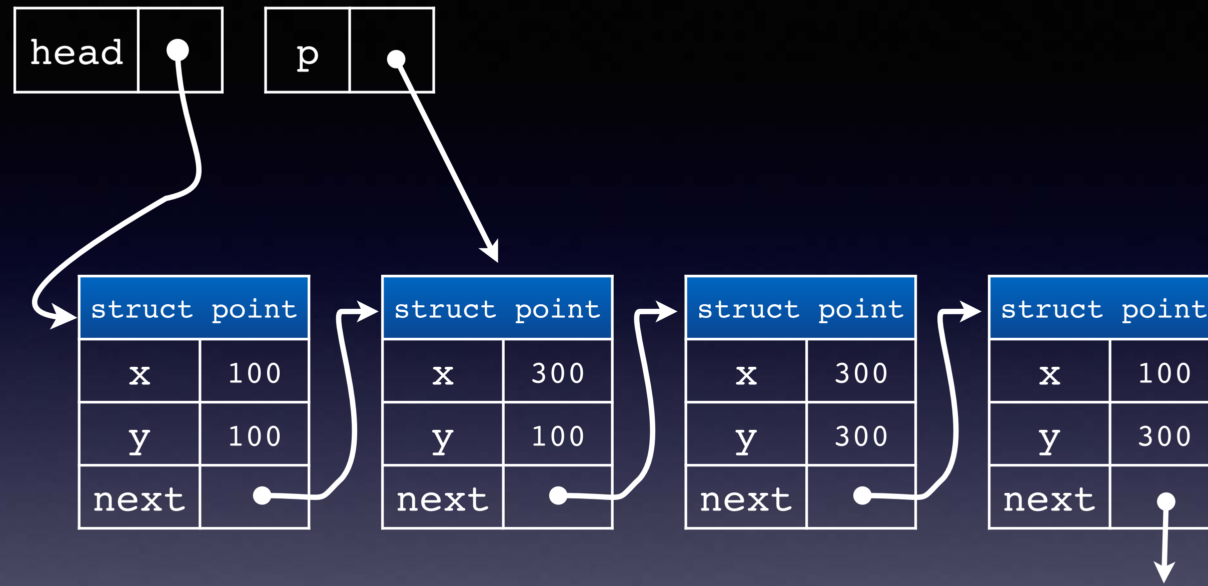
p or whatever variable name you use in the algorithm on the previous slide



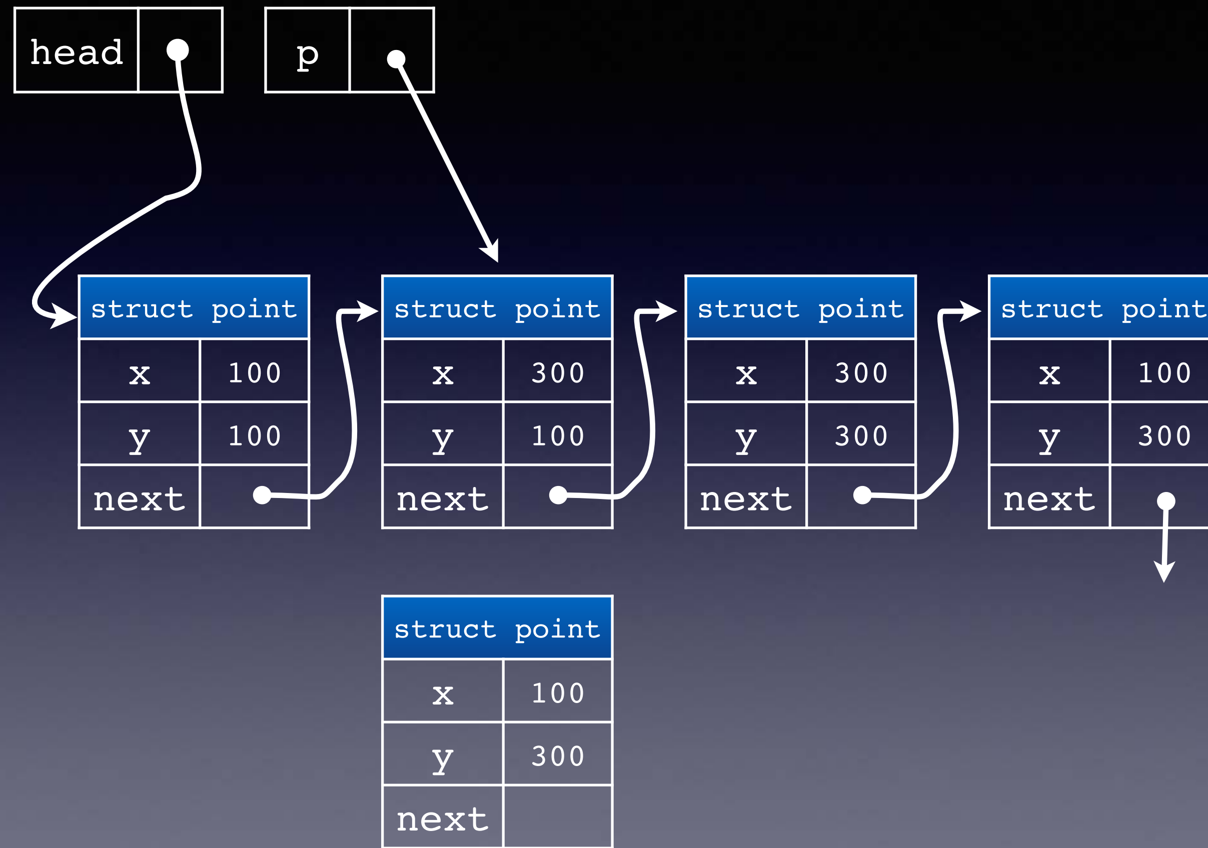
head
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next
struct point
x
100
y
300
next

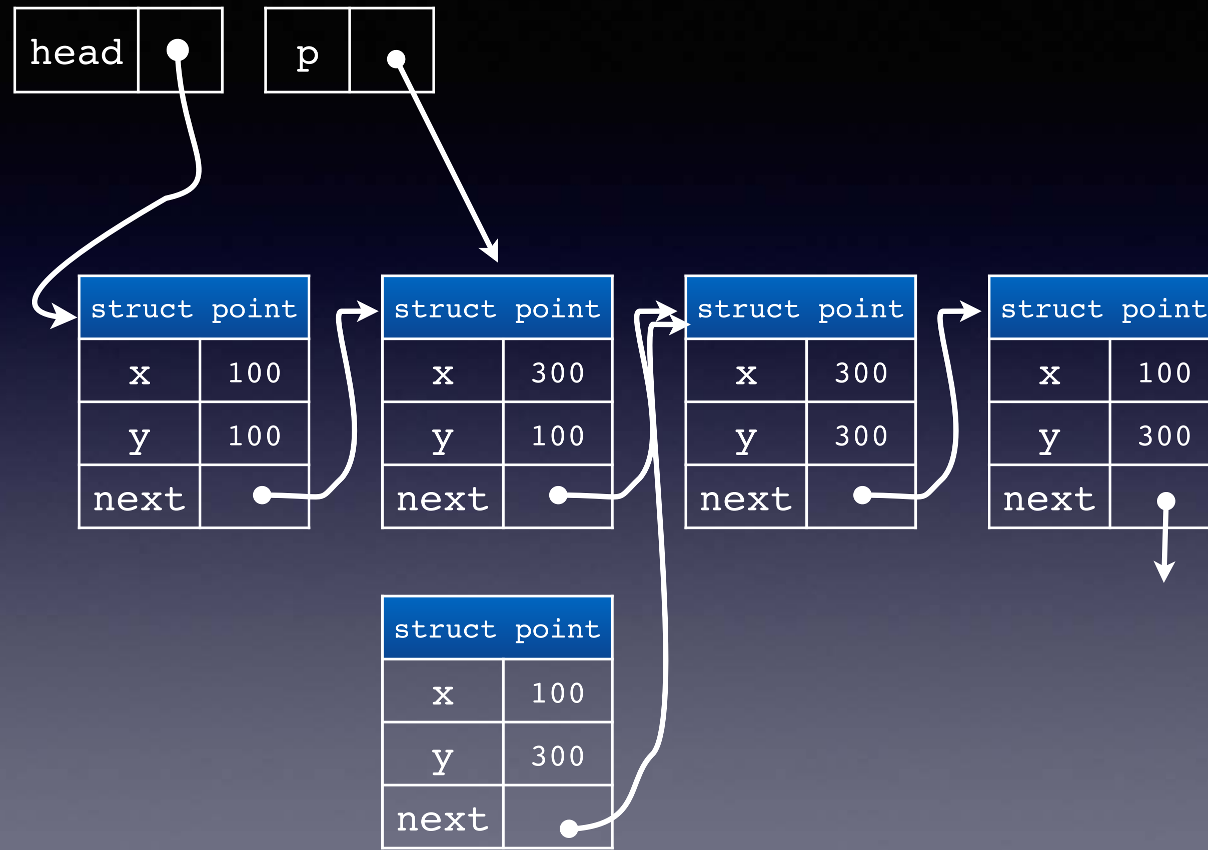
head
p

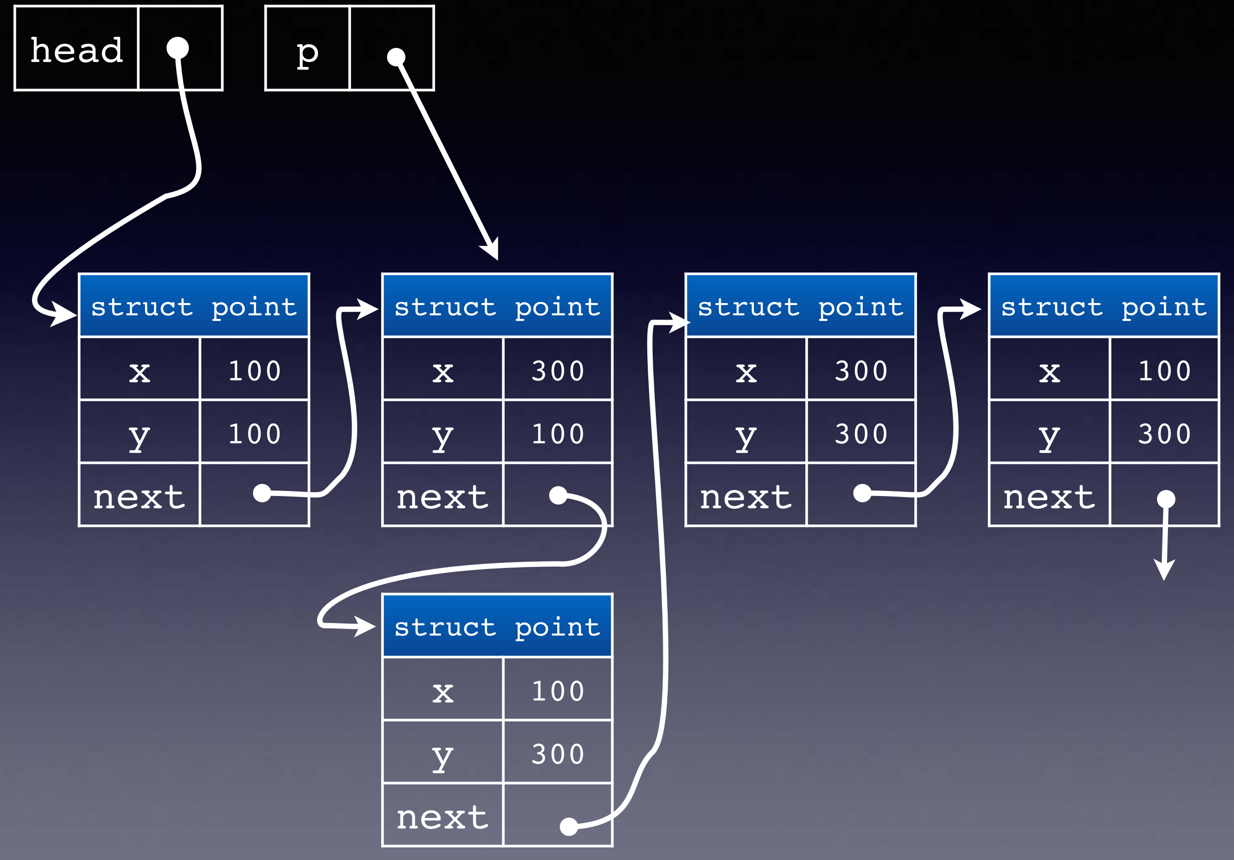

struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next
struct point
x
100
y
300
next

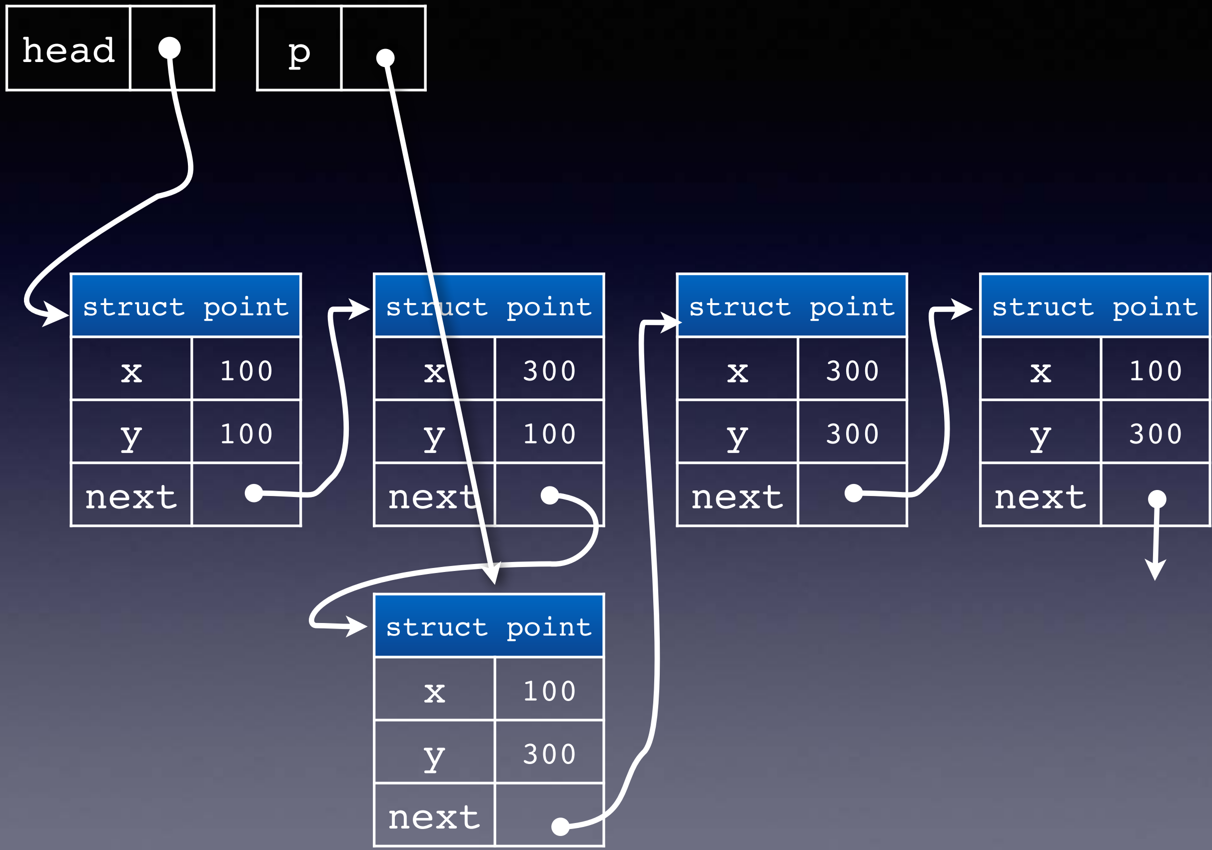


head
p
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next
struct point
x
100
y
300
next


head
p
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next
struct point
x
100
y
300
next
struct point
x
100
y
300
next


head
p
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next
struct point
x
100
y
300
next
struct point
x
100
y
300
next


head
p
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next
struct point
x
100
y
300
next
struct point
x
100
y
300
next



head
p
struct point
x
100
y
100
next
struct point
x
300
y
100
next
struct point
x
300
y
300
next
struct point
x
100
y
300
next
struct point
x
100
y
300
next

`while` or `for`

- So far, examples have used `while` loops
- But we could also use a `for` loop
- Our initial statement sets `p` to equal `head`
- Condition is `p!=NULL` or `p->next!=NULL`
- Our step is `p = p->next`
- Will encounter both in real programs

```
struct point *head;
struct point *p;
…

p = head;
while(p != NULL)
{
   /* Do stuff */
   p = p->next;
}

for(p = head; p != NULL; p = p->next;)
{
   /* Do stuff */
}
```

These are equivalent

Advanced Linked Lists

- Doubly-Linked list — points to the next *and* previous items in the list
- Allows you to traverse the list in both directions
- Possible to remove the need to test whether `head` is `NULL`