Linked Lists

Toolkit Implementations

list_length

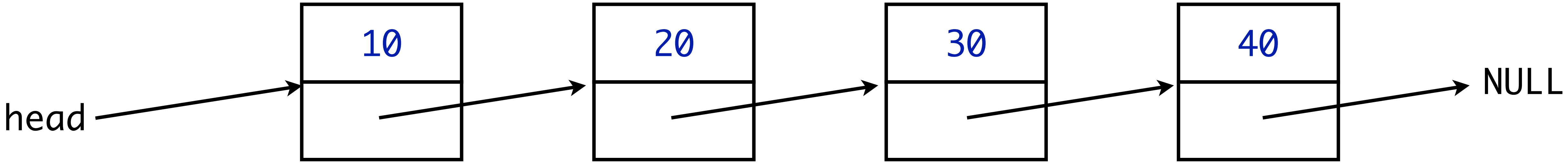
Determine the length of a linked list:

```
// returns the number of nodes in a linked list
size_t list_length(const Node* head_ptr);

// precondition:
//   head_ptr is the head pointer of a linked list
// postcondition:
//   the value returned is the number of nodes in the
//   linked list
```

list_length

How would you find the size of this list?

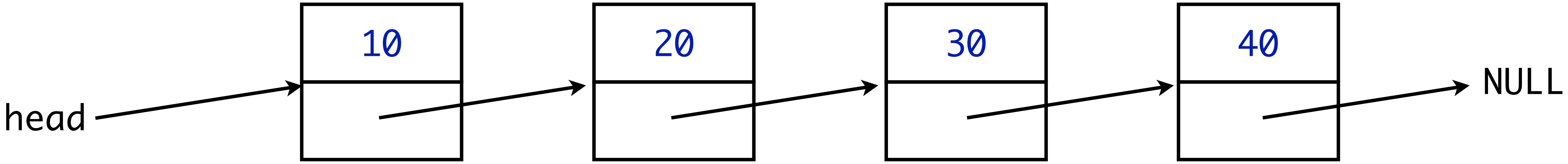


Traverse the entire list...

- count each node one at a time
- must return 0 for an empty list

list_length

How would you find the size of this list?

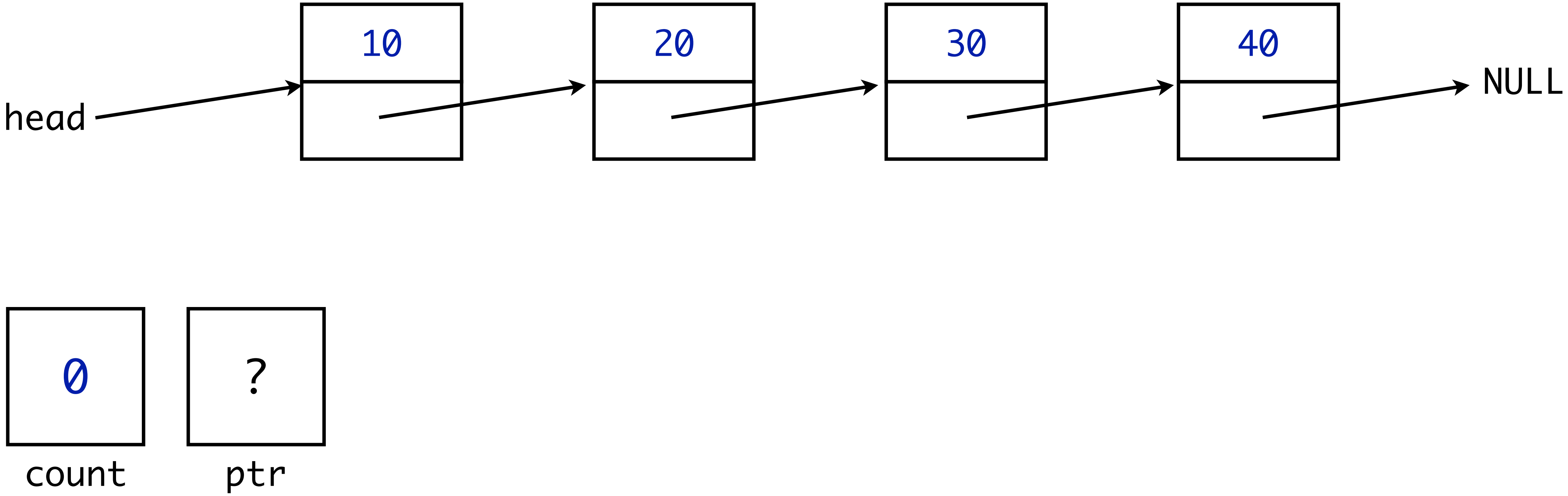


Traverse the entire list...

```
ptr = head;          // start at the beginning
ptr != NULL;         // are we at the end of the list?
ptr = ptr->link();   // advance to next node
```

list_length

How would you find the size of this list?

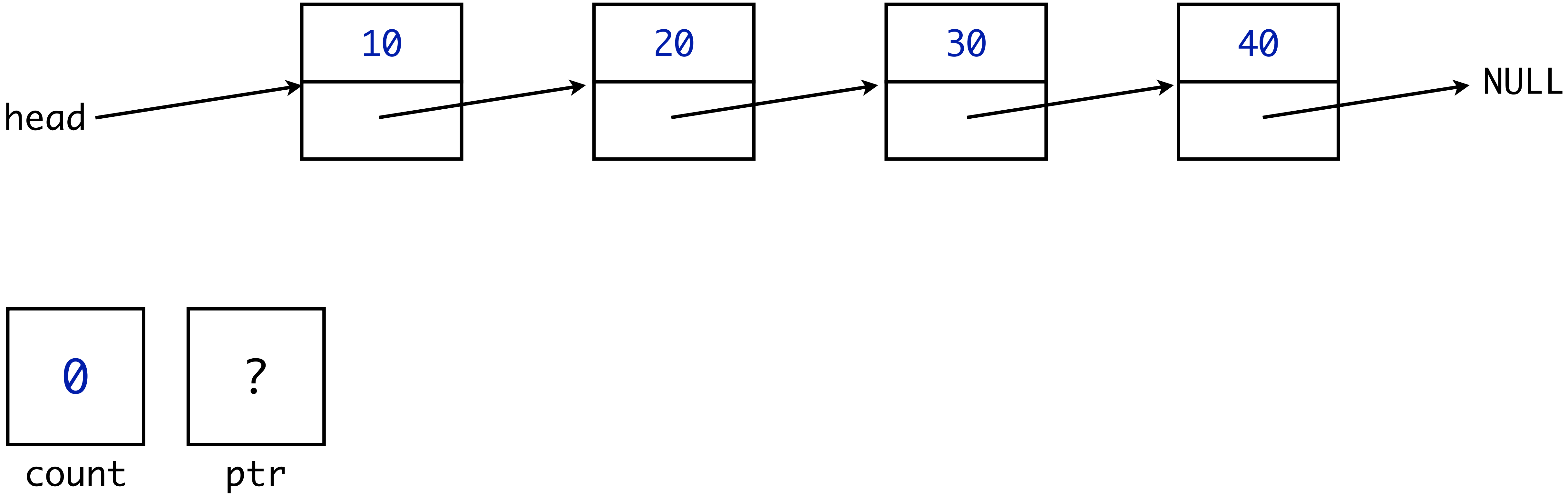


Track the number of nodes and current position:

```
size_t count = 0;  // number of nodes
const Node* ptr;   // current position
```

list_length

How would you find the size of this list?

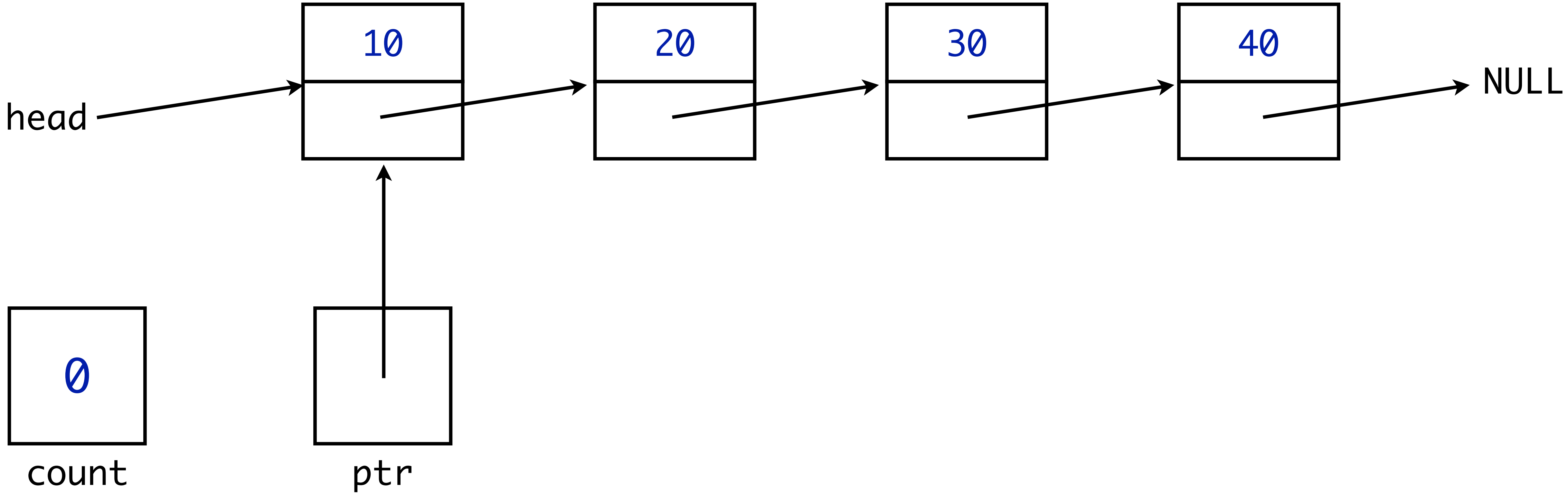


Traverse the list and count each node:

```
for (ptr = head; ptr != NULL; ptr = ptr->link()) {
    count++;
}
```

list_length

How would you find the size of this list?

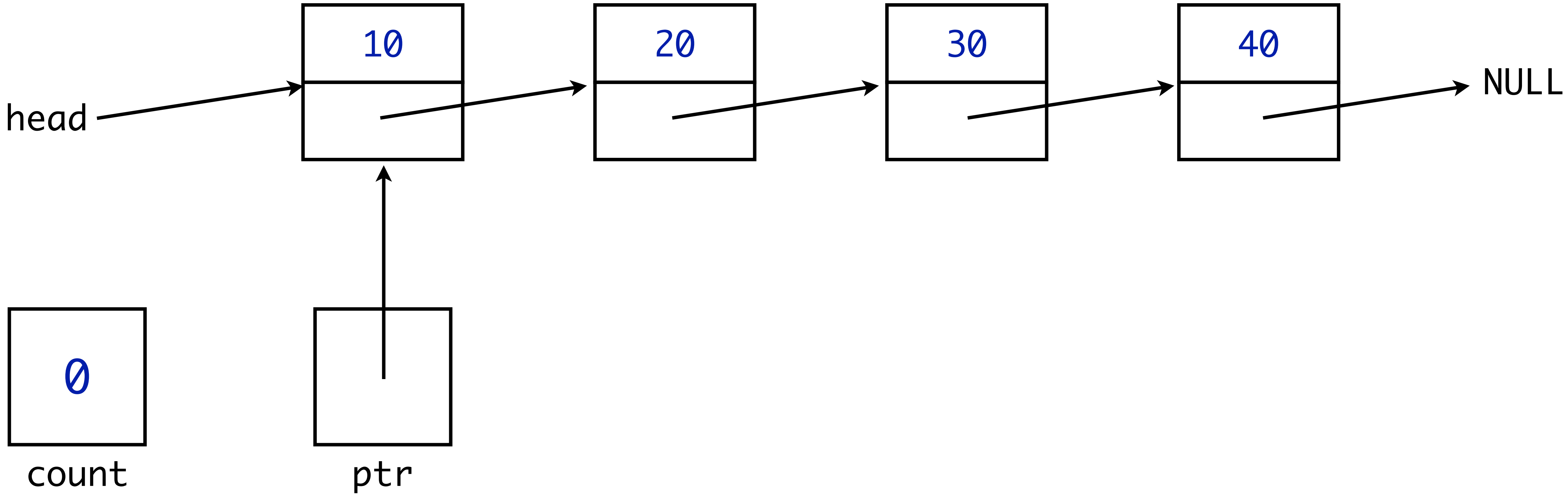


Start at the beginning of the list...

```
for (ptr = head; ptr != NULL; ptr = ptr->link()) {
    count++;
}
```

list_length

How would you find the size of this list?

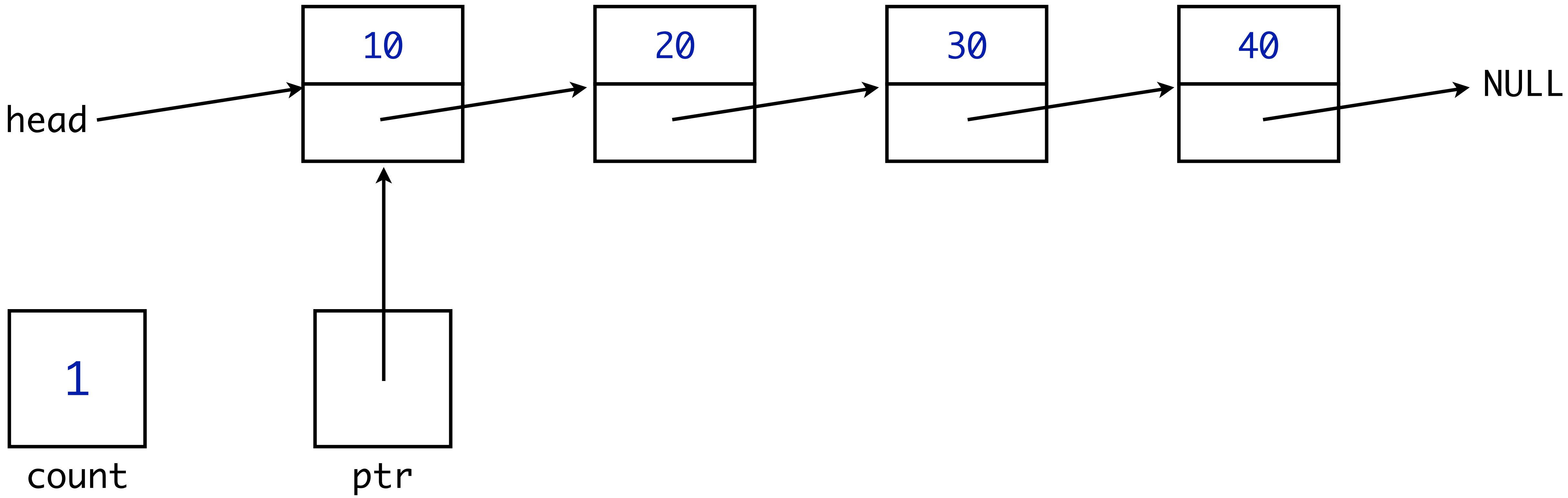


Check that we're not at the end yet...

```
for (ptr = head; ptr != NULL; ptr = ptr->link()) {
    count++;
}
```

list_length

How would you find the size of this list?

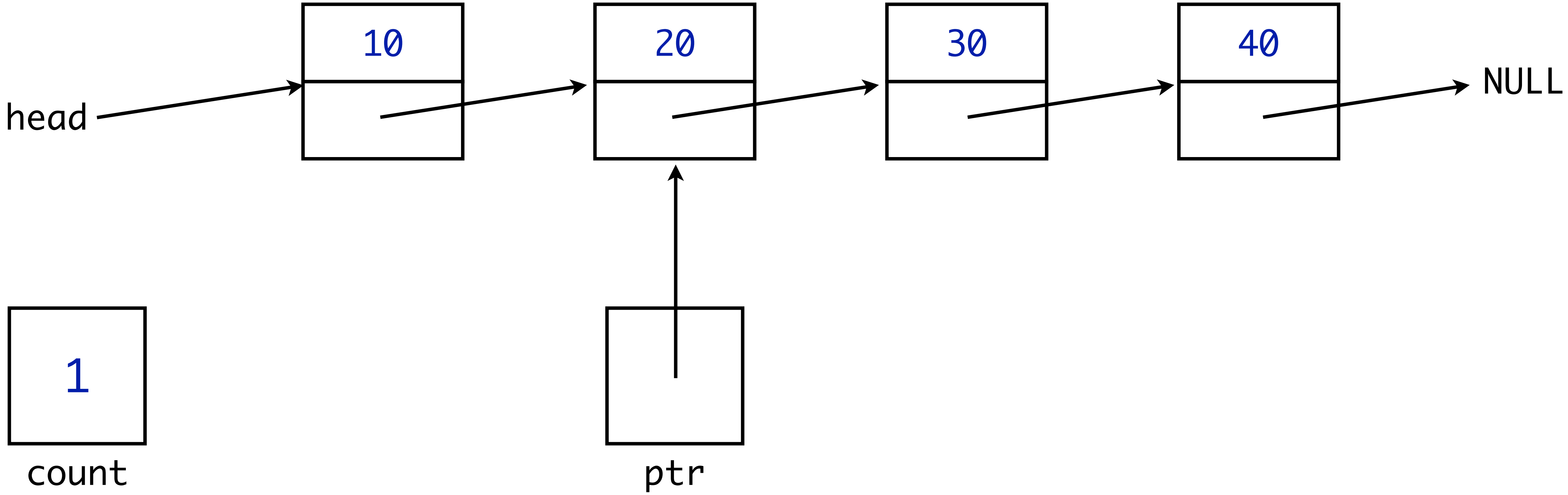


The pointer is valid, so count the node...

```
for (ptr = head; ptr != NULL; ptr = ptr->link()) {
    count++;
}
```

list_length

How would you find the size of this list?

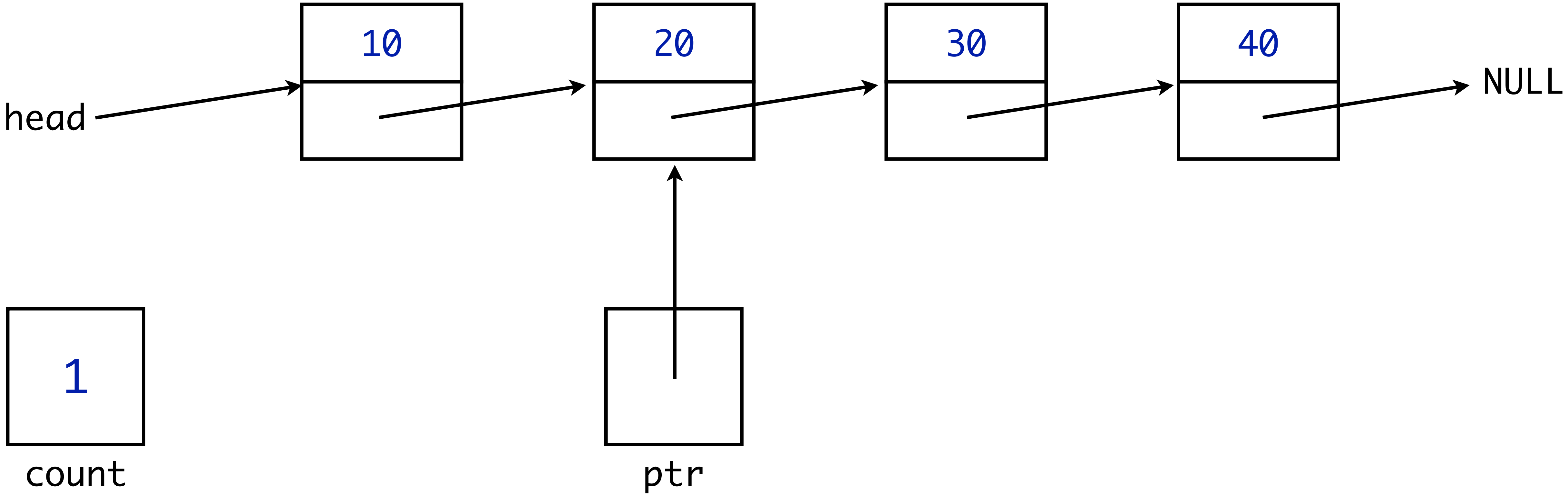


Advance the pointer to the next node...

```
    for (ptr = head; ptr != NULL; ptr = ptr->link()) {
        count++;
    }
```

list_length

How would you find the size of this list?

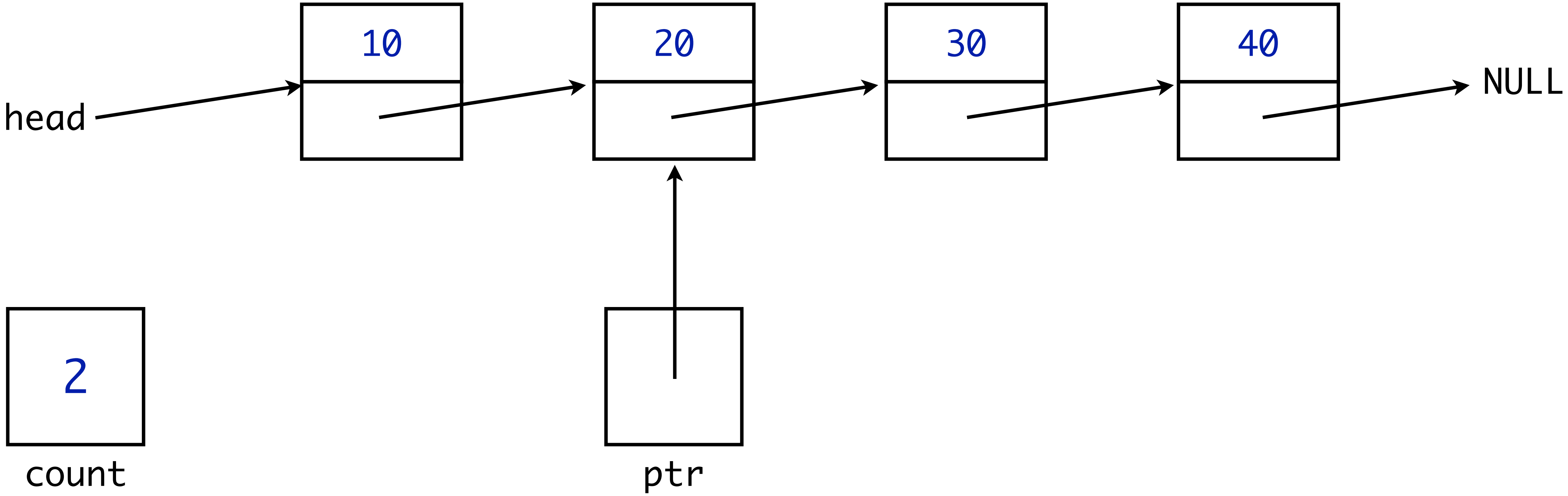


Check that we're not at the end yet...

```
for (ptr = head; ptr != NULL; ptr = ptr->link()) {
    count++;
}
```

list_length

How would you find the size of this list?

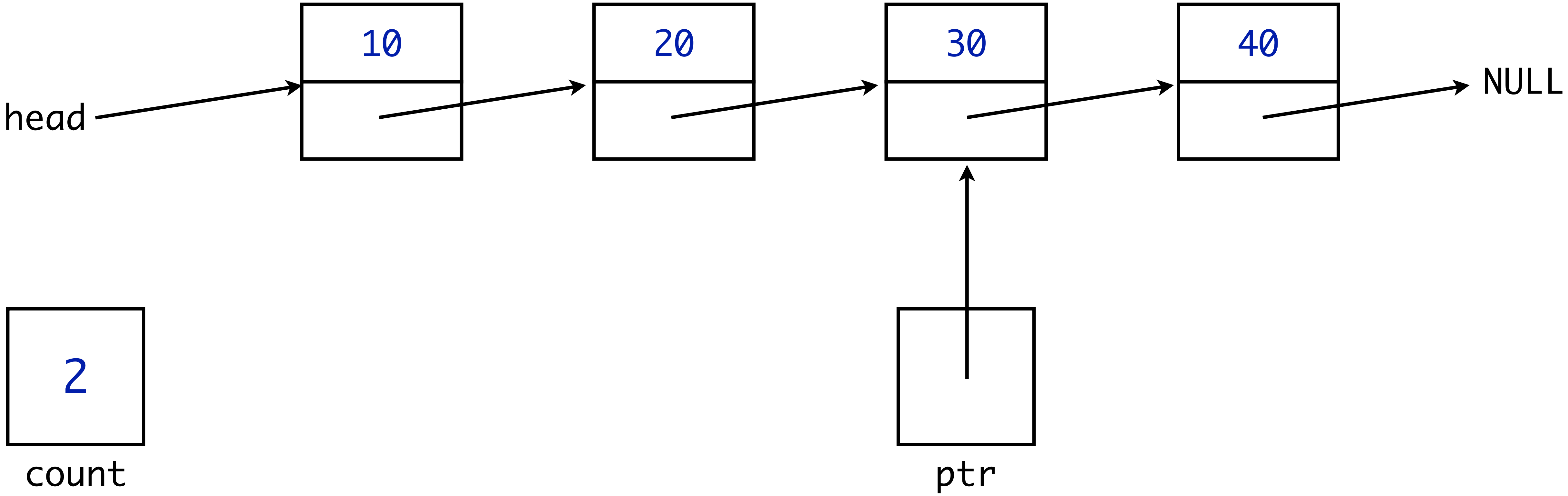


The pointer is valid, so count the node...

```
for (ptr = head; ptr != NULL; ptr = ptr->link()) {
    count++;
}
```

list_length

How would you find the size of this list?

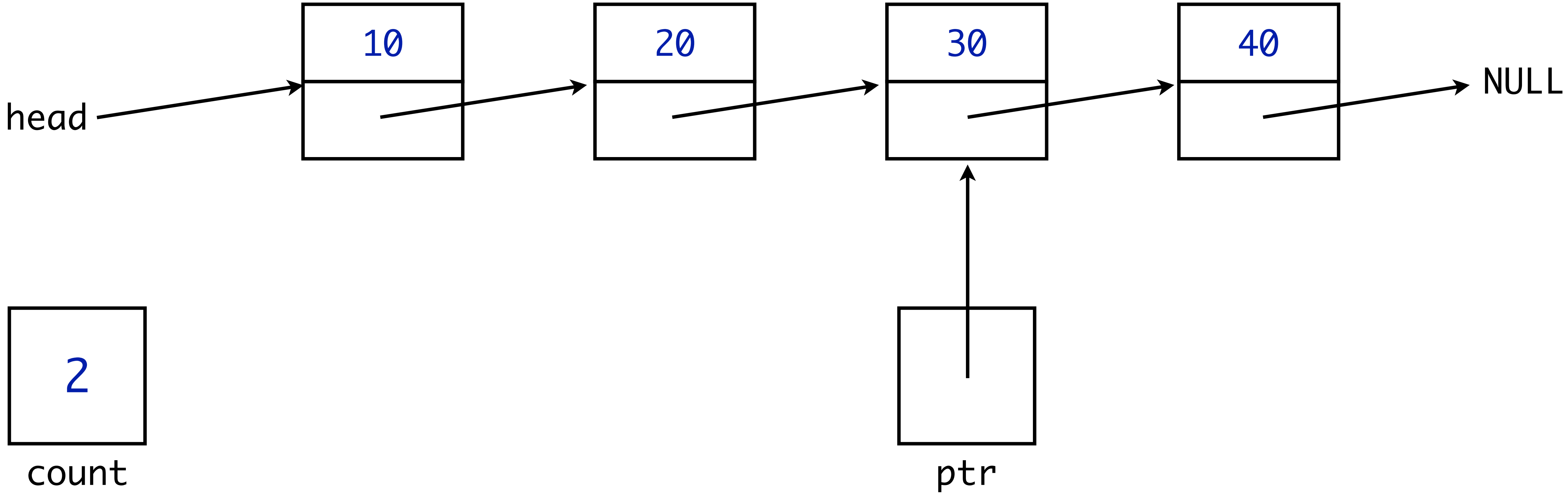


Advance the pointer to the next node...

```
for (ptr = head; ptr != NULL; ptr = ptr->link()) {
    count++;
}
```

list_length

How would you find the size of this list?

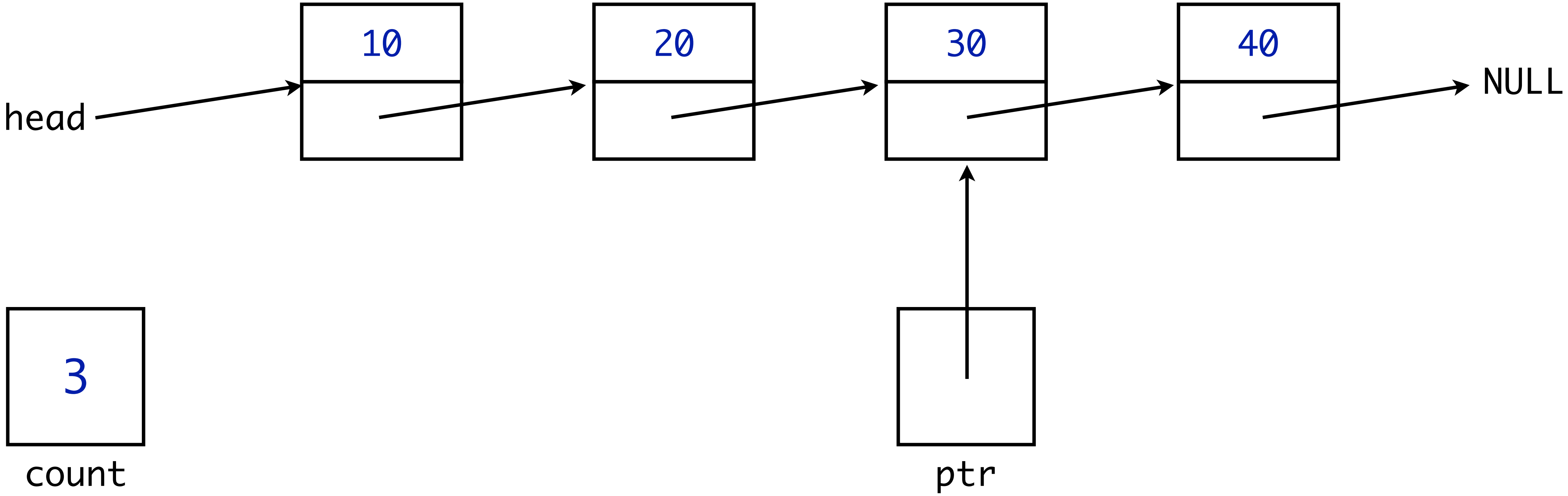


Check that we're not at the end yet...

```
for (ptr = head; ptr != NULL; ptr = ptr->link()) {
    count++;
}
```

list_length

How would you find the size of this list?

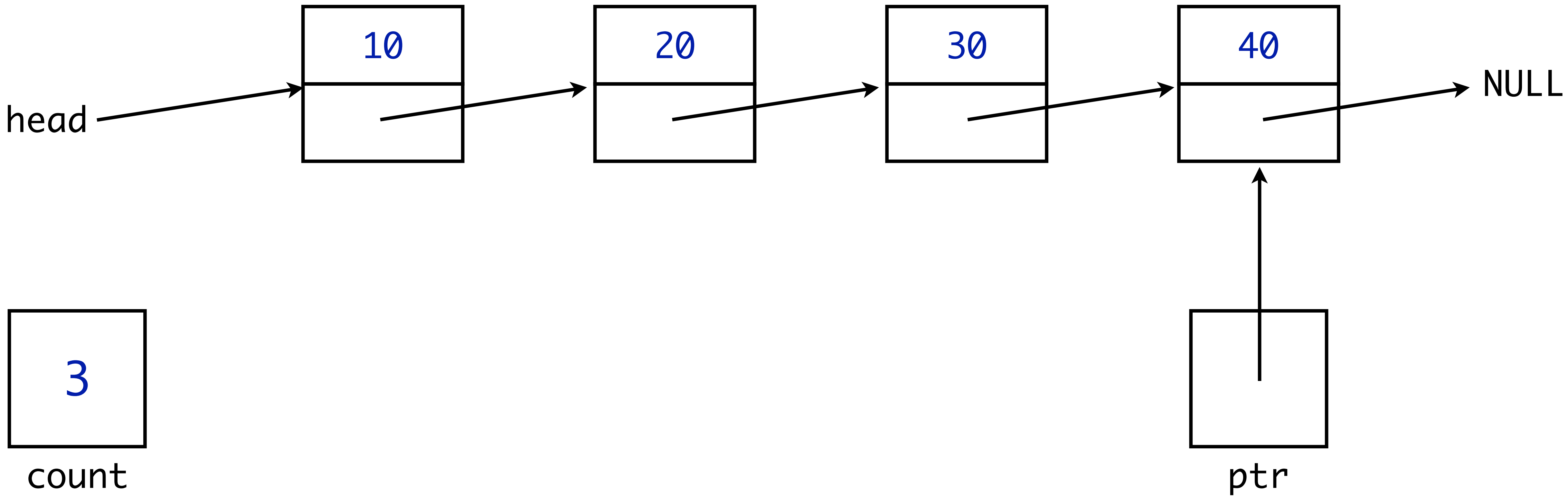


The pointer is valid, so count the node...

```
for (ptr = head; ptr != NULL; ptr = ptr->link()) {
    count++;
}
```

list_length

How would you find the size of this list?

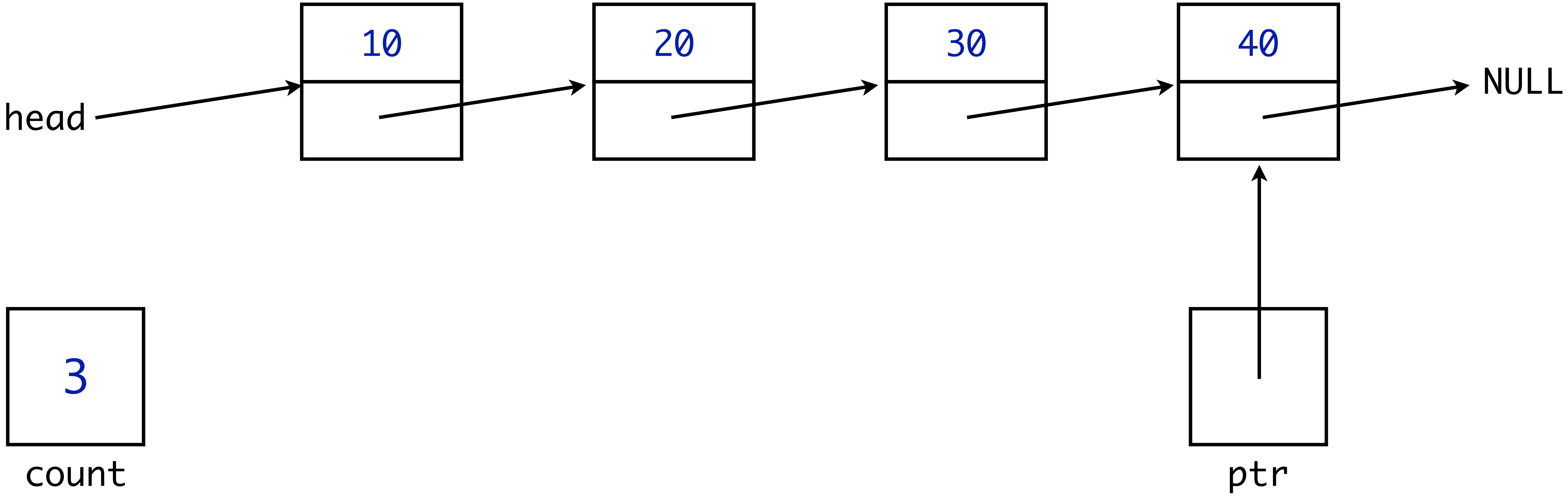


Advance the pointer to the next node...

```
    for (ptr = head; ptr != NULL; ptr = ptr->link()) {
        count++;
    }
```

list_length

How would you find the size of this list?

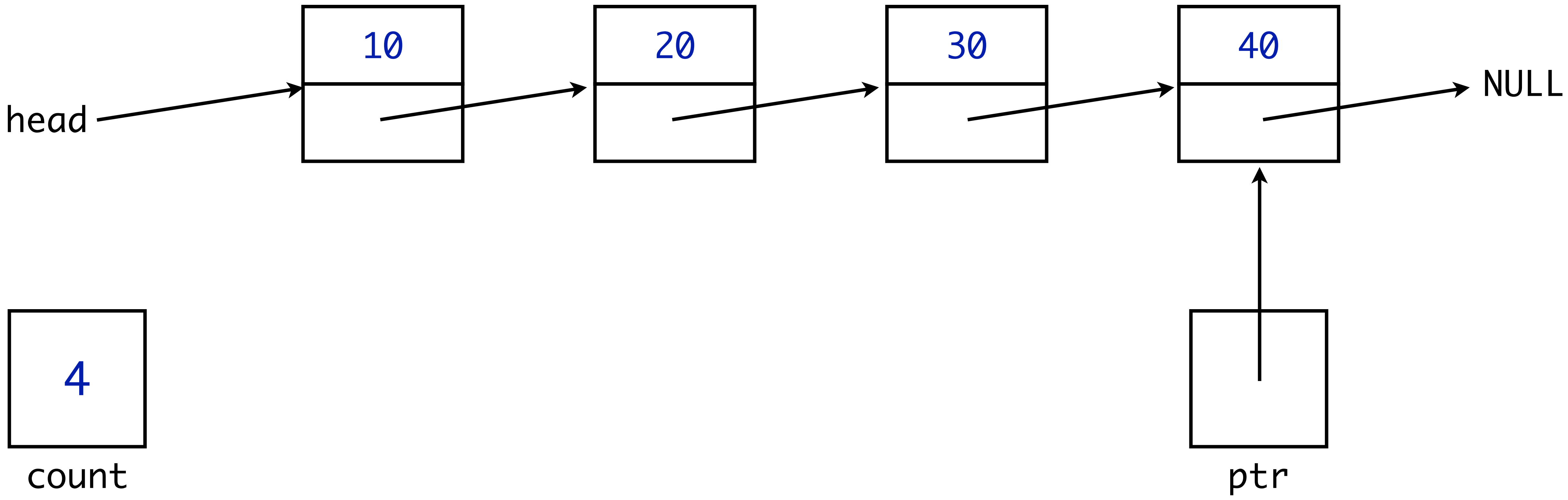


Check that we're not at the end yet...

```
for (ptr = head; ptr != NULL; ptr = ptr->link()) {
    count++;
}
```

list_length

How would you find the size of this list?

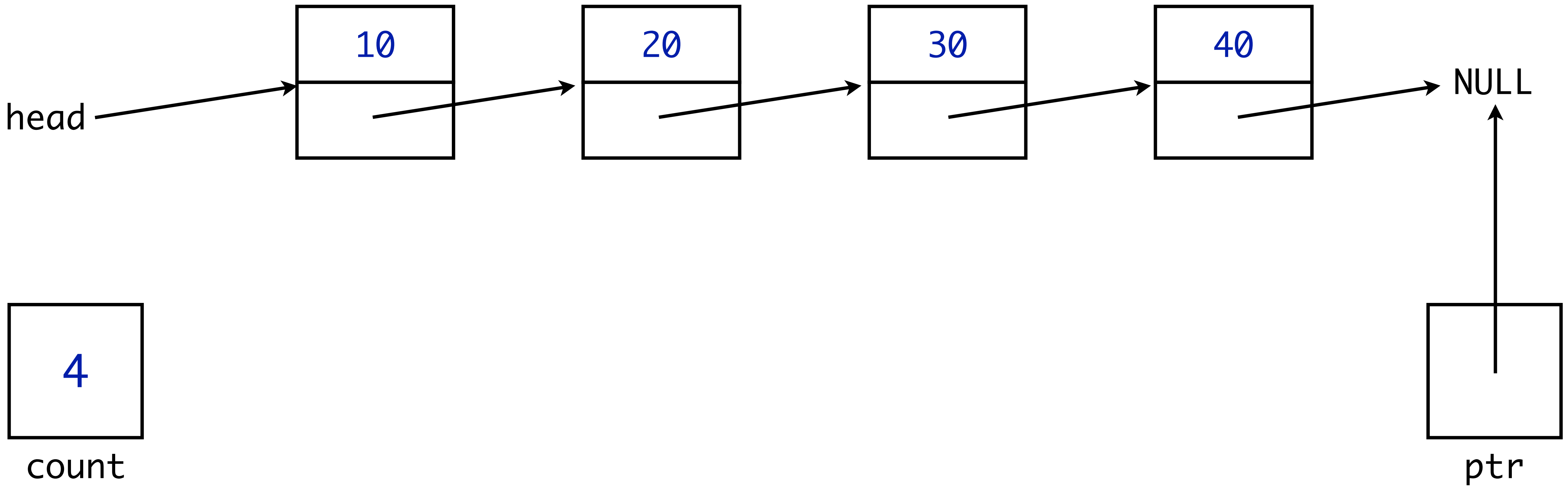


The pointer is valid, so count the node...

```
for (ptr = head; ptr != NULL; ptr = ptr->link()) {
    count++;
}
```

list_length

How would you find the size of this list?

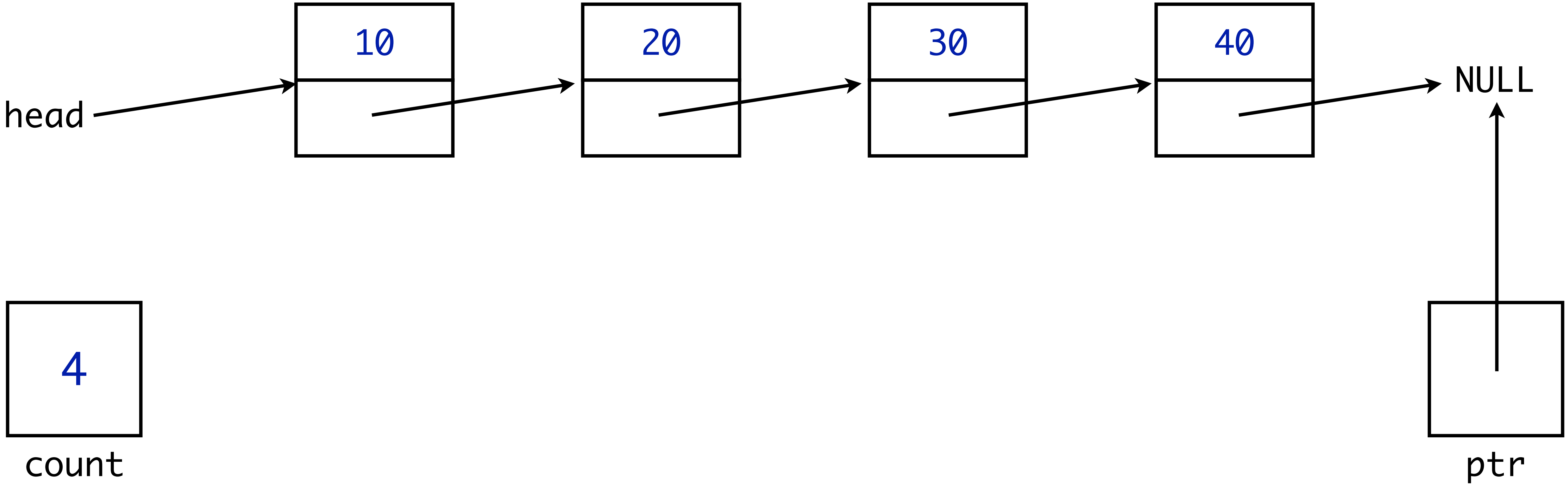


Advance the pointer to the next node...

```
    for (ptr = head; ptr != NULL; ptr = ptr->link()) {
        count++;
    }
```

list_length

How would you find the size of this list?

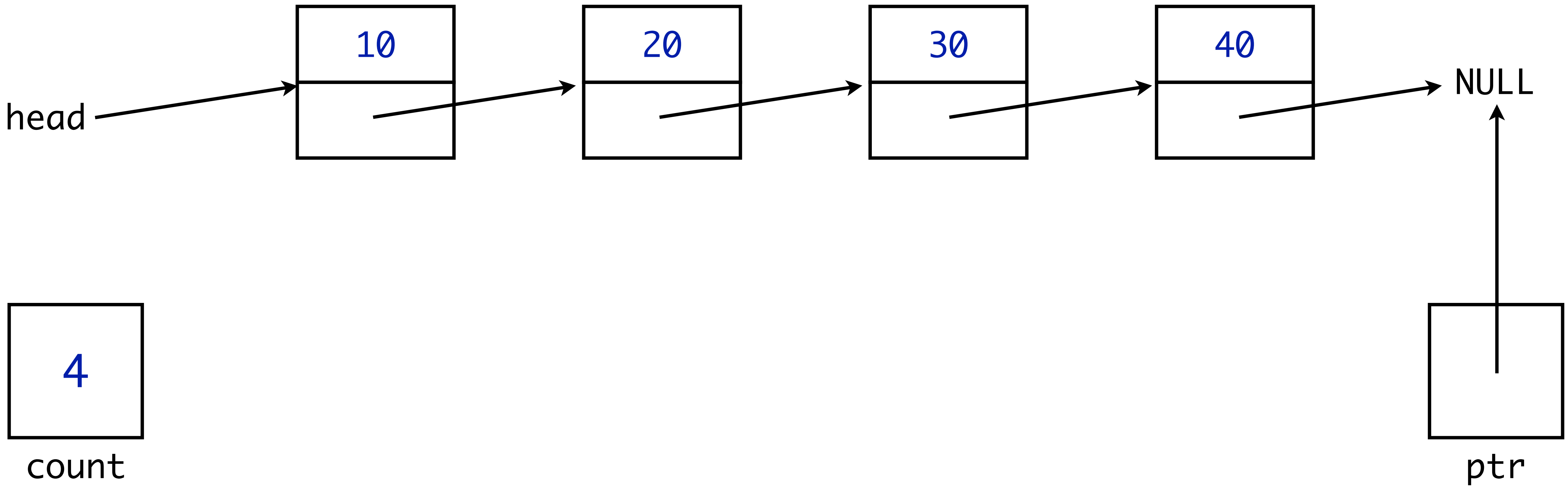


Check that we're not at the end yet...

```
for (ptr = head; ptr != NULL; ptr = ptr->link()) {
    count++;
}
```

list_length

How would you find the size of this list?



We've at the end of the list, so all done!

```
    for (ptr = head; ptr != NULL; ptr = ptr->link()) {
        count++;
    }
```

list_length

Determine the length of a linked list:

```
    // returns the number of nodes in a linked list
    size_t list_length(const Node* head_ptr) {
        size_t count = 0;
        const Node* ptr;

        for (ptr = head_ptr; ptr != NULL; ptr = ptr->link())
            count++;

        return count;
    }
```

list_head_insert

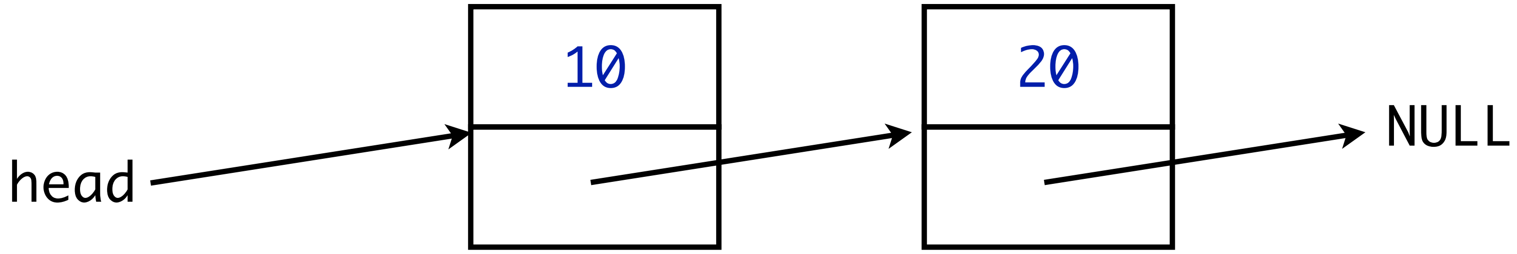
Insert an item at the front of a list:

```
// inserts @entry at the beginning of @head_ptr’s list
void list_head_insert(Node*& head_ptr,
                      const Node::value_type& entry);


// precondition:
//   head_ptr is the head pointer of a linked list
// postcondition:
//   a new node containing the given entry has been
//   added at the head of the linked list; head_ptr now
//   points to the head of the new, longer linked list
```

list_head_insert

How would you insert 5 at the start of this list?

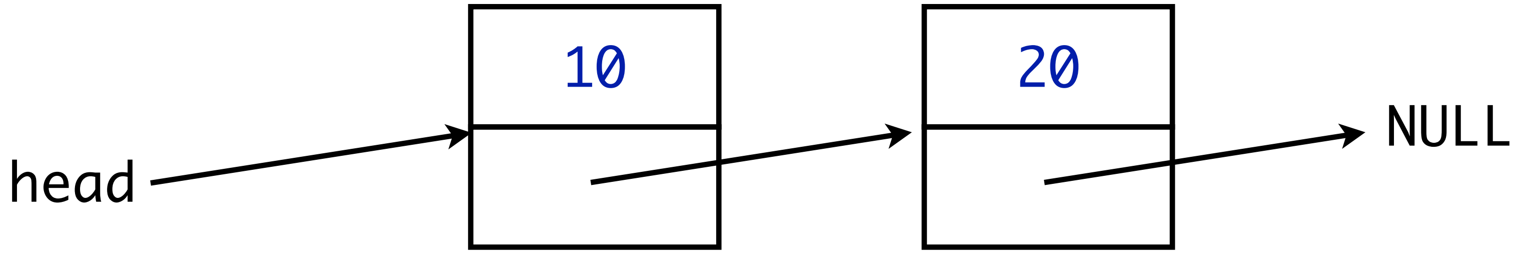


We need to do a couple of tasks:

- create a new node and set its data field to 5
- set the new node to point to the old head node (or NULL, if list is empty)
- update the head pointer to point to the new node

list_head_insert

How would you insert 5 at the start of this list?

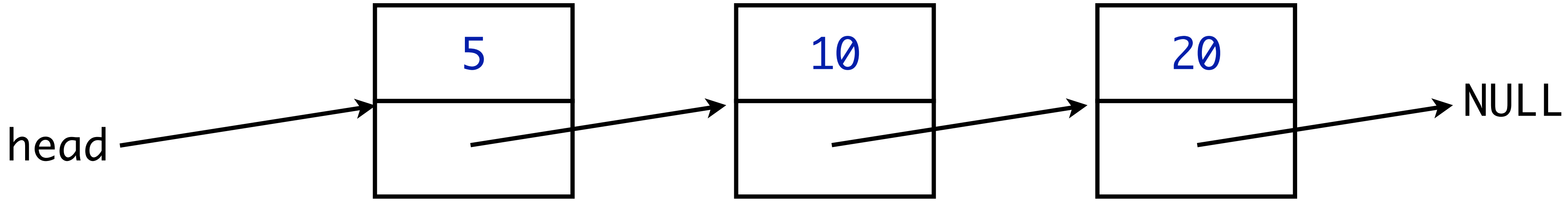


Remember, the node constructor looks like this:

```
// create a node with data @d and link @n
Node(const value_type& d = value_type(),
     Node* n = NULL): data(d), next(n) { }
```

list_head_insert

How would you insert 5 at the start of this list?

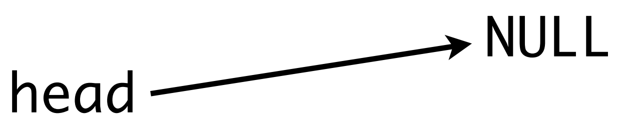


This one line of code does everything we need:

```
// insert a new node at the head of the list
head = new Node(5, head);
```

list_head_insert

How would you insert 5 at the start of this list?



NULL

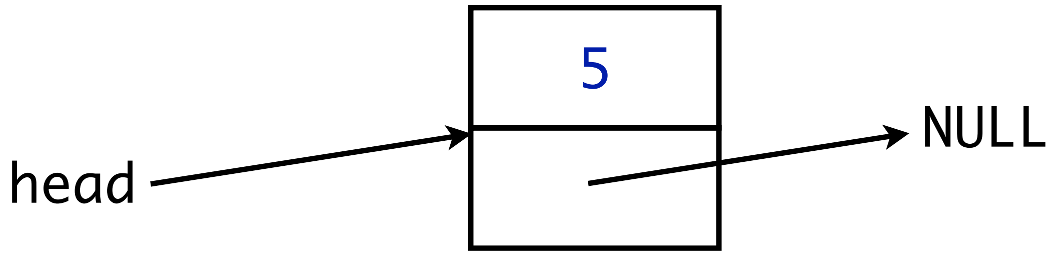
head

It works even if the list is initially empty:

```
// insert a new node at the head of the list
head = new Node(5, head);
```

list_head_insert

How would you insert 5 at the start of this list?



It works even if the list is initially empty:

```
// insert a new node at the head of the list
head = new Node(5, head);
```

list_head_insert

Insert an item at the front of a list:

```
// inserts @entry at the beginning of @head_ptr’s list
void list_head_insert(Node*& head_ptr,
                      const Node::value_type& entry)
{
    head_ptr = new Node(entry, head_ptr);
}
```

list_insert

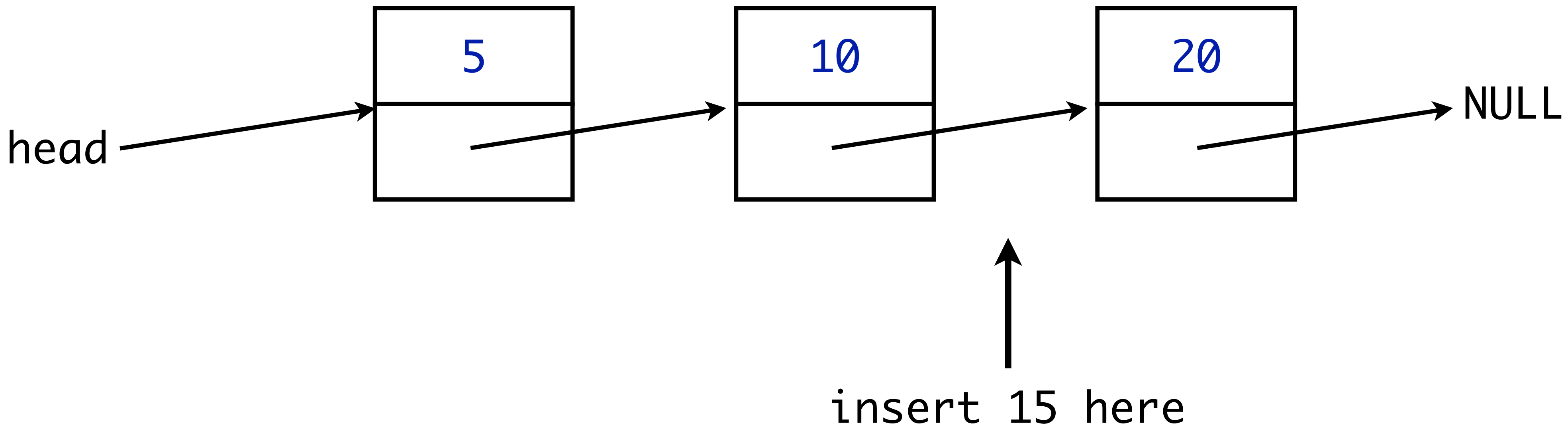
Insert an item after a node in a list:

```
// inserts @entry after @previous_ptr in a list
void list_insert(Node* previous_ptr,
                 const Node::value_type& entry);


// precondition:
//   previous_ptr points to a node in a linked list
// postcondition:
//   a new node containing the given entry has been
//   added after the node pointed at by previous_ptr
```

list_insert

How would you insert 15 between 10 and 20?

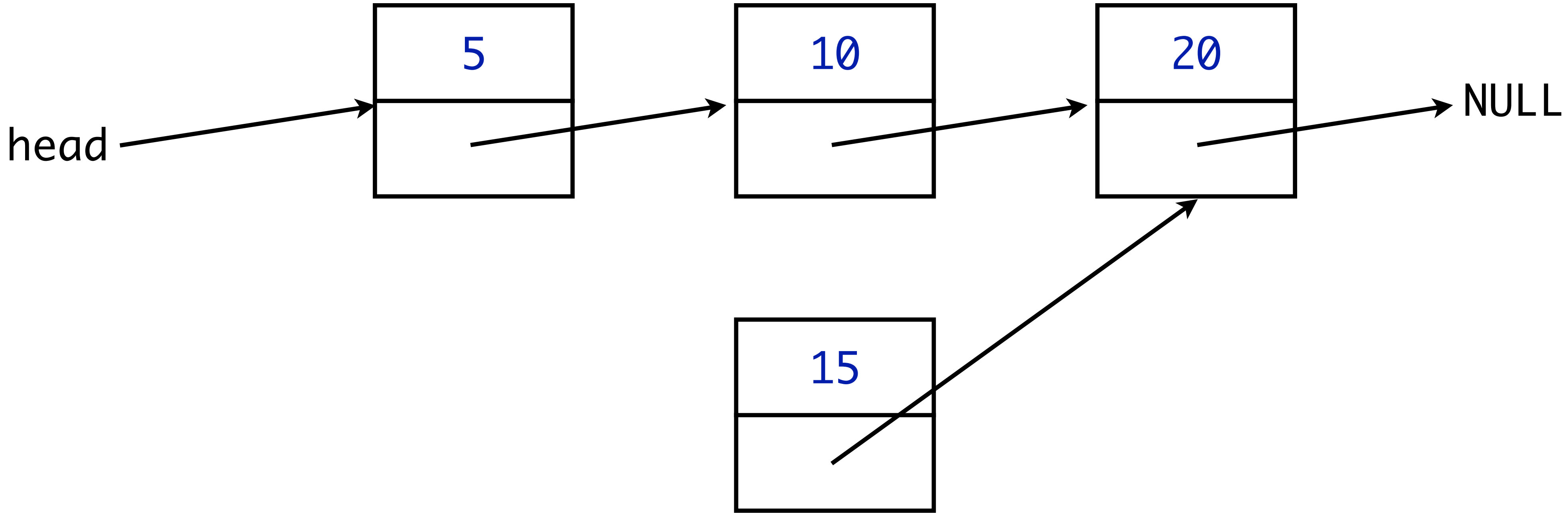


We need a pointer to the previous node...

```
// the node just before the insert location
Node* prev_node;
```

list_insert

How would you insert 15 between 10 and 20?

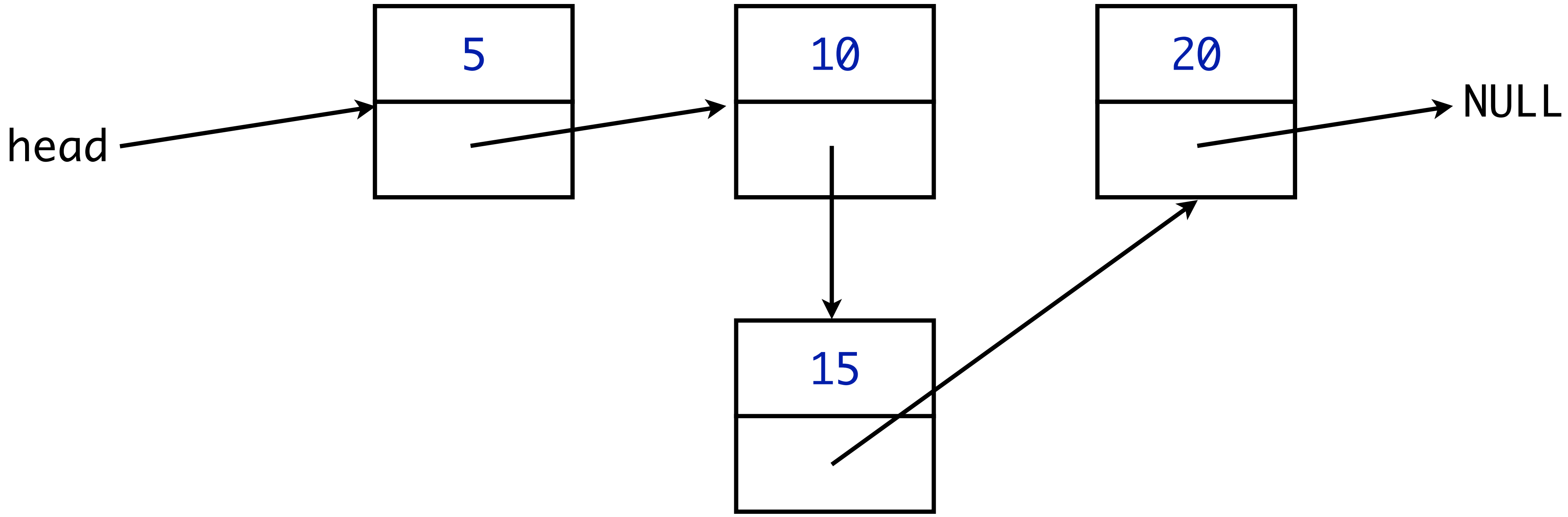


Then we need to create the new node:

```
// the node to be inserted
Node* new_node = new Node(15, prev_node->link());
```

list_insert

How would you insert 15 between 10 and 20?

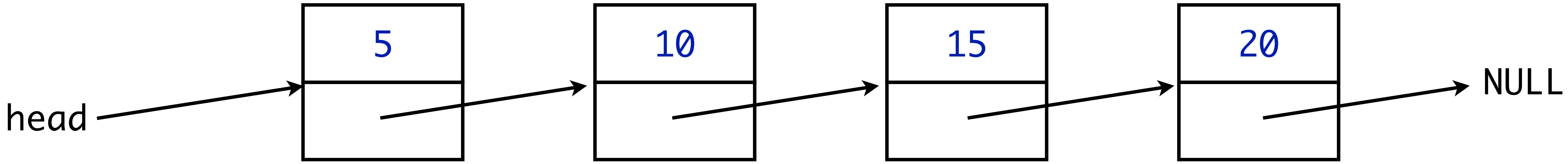


Then update the previous node’s link:

```
// update the previous node to point to the new one
prev_node->set_link(new_node);
```

list_insert

How would you insert 15 between 10 and 20?



We can do all that with one line of code:

```
// insert a new node after prev_node
prev_node->set_link(new Node(15, prev_node->link()));
```

list_insert

Insert an item after a node in a list:

```
// inserts @entry after @previous_ptr in a list
void list_insert(Node* previous_ptr,
                 const Node::value_type& entry)
{
    previous_ptr->set_link(
        new Node(entry, previous_ptr->link())
    );
}
```

list_search

Search for an item in a list (non-const version):

```
// returns a pointer to @target if it’s in a linked list
Node* list_search(Node* head_ptr,
                  const Node::value_type& target);


// precondition:
//   head_ptr is the head pointer of a linked list
// postcondition:
//   the pointer returned points to the first node
//   containing the specified target in its data member.
//   If there is no such node, NULL is returned
```

list_search

Search for an item in a list (non-const version):

```
// returns a pointer to @target if it’s in a linked list
Node* list_search(Node* head_ptr,
                  const Node::value_type& target)
{
    Node* n;

    for (n = head_ptr; n != NULL; n = n->link())
        if (n->data() == target) return n;

    return NULL;
}
```

list_search

Search for an item in a list (const version):

```
// returns a pointer to @target if it’s in a linked list
const Node* list_search(const Node* head_ptr,
                        const Node::value_type& target);


// precondition:
//   head_ptr is the head pointer of a linked list
// postcondition:
//   the pointer returned points to the first node
//   containing the specified target in its data member.
//   If there is no such node, NULL is returned
```

list_search

Search for an item in a list (const version):

```
// returns a pointer to @target if it’s in a linked list
const Node* list_search(const Node* head_ptr,
                        const Node::value_type& target)
{
    const Node* n;

    for (n = head_ptr; n != NULL; n = n->link())
        if (n->data() == target) return n;

    return NULL;
}
```

list_locate

Search for an item at a specific location in a list (non-const version):

```
// returns the item at @position in a linked list
Node* list_locate(Node* head_ptr,
                  size_t position);

// precondition:
//   head_ptr is the head pointer of a linked list, and
//   position is greater than 0
// postcondition:
//   the pointer returned points to the node at the
//   specified position in the list (starting at 1). If
//   there is no such position, then NULL is returned
```

list_locate

Search for an item at a specific location in a list (non-const version):

```
// returns the item at @position in a linked list
Node* list_locate(Node* head_ptr,
                  size_t position)
{
    Node* n = head_ptr;

    for (size_t i = 1; i < position && n != NULL; i++)
        n = n->link();

    return n;
}
```

list_locate

Search for an item at a specific location in a list (const version):

```
// returns the item at @position in a linked list
const Node* list_locate(const Node* head_ptr,
                        size_t position);


// precondition:
//   head_ptr is the head pointer of a linked list, and
//   position is greater than 0
// postcondition:
//   the pointer returned points to the node at the
//   specified position in the list (starting at 1). If
//   there is no such position, then NULL is returned
```

list_locate

Search for an item at a specific location in a list (const version):

```
// returns the item at @position in a linked list
const Node* list_locate(const Node* head_ptr,
                        size_t position)
{
    const Node* n = head_ptr;

    for (size_t i = 1; i < position && n != NULL; i++)
        n = n->link();

    return n;
}
```

list_head_remove

Removes the node at the head of a list:

```
// removes the node at the head of a linked list
void list_head_remove(Node*& head_ptr);

// precondition:
//   head_ptr is the head pointer of a linked list, with
//   at least one node
// postcondition:
//   the head node has been removed and returned to the
//   heap; head_ptr is now the head pointer of the new,
//   shorter linked list
```

list_head_remove

Removes the node at the head of a list:

```
// removes the node at the head of a linked list
void list_head_remove(Node*& head_ptr) {
    Node* remove_ptr = head_ptr;

    head_ptr = head_ptr->link();

    delete remove_ptr;
}
```

list_remove

Removes the node after the specified node:

```
// removes the node following @previous_ptr in a list
void list_remove(Node* previous_ptr);

// precondition:
//   previous_ptr points to a node in a linked list and
//   is not the tail node of the list
// postcondition:
//   the node after previous_ptr has been removed from
//   the linked list
```

list_remove

Removes the node after the specified node:

```
// removes the node following @previous_ptr in a list
void list_remove(Node* previous_ptr) {
    Node* remove_ptr = previous_ptr->link();


    previous_ptr->set_link( remove_ptr->link() );


    delete remove_ptr;
}
```

list_clear

Clears the linked list:

```
// clears the linked list identified by @head_ptr
void list_clear(Node*& head_ptr);

// precondition:
//   head_ptr is the head pointer of a linked list
// postcondition:
//   all nodes of the list have been returned to the
//   heap, and the head_ptr is now NULL
```

list_clear

Clears the linked list:

```
// clears the linked list identified by @head_ptr
void list_clear(Node*& head_ptr) {
    while (head_ptr != NULL) {
        list_head_remove(head_ptr);
    }
}
```