Linked Lists 2

Elyse Cornwall

August 1, 2023

Contributions made from previous CS106B Instructors

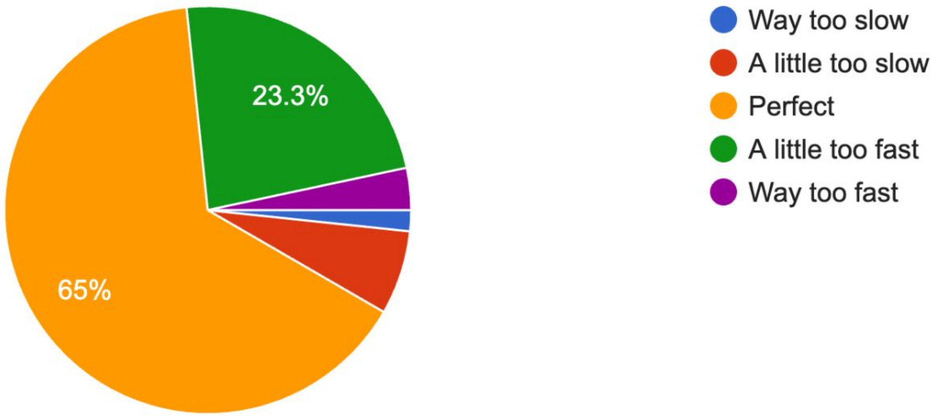
Announcements

- Change of grading basis deadline is this Friday at 5pm PT
  - Come chat with us (or check out this resource) if you're considering whether to take for letter grade or credit/no credit

Feedback

Rate the pace of lecture

60 responses



Feedback

Things you liked:

"I really like the **drawings on the board** as it provides a different, more **visual method of learning**"

"**Going through the code slower** is helping a lot"

"**office hours!**"

"The explanations that are done in a very **step-by-step** way, with each slide incrementing one change has helped make concepts very clear."

"The provided code during lecture **helps a lot to start the assignments**"

Feedback

Places we can improve:

“it's helpful to **recap multiple times** in between what bigger picture it fits into… (as opposed to one big recap in the end)”

“I think it would be helpful to have **more interactive stuff** during lecture (trying to code on our own, answering practice questions, discussing with others)”

“When there is 5m of class left, **not to quickly rush through the last slides**”

“I spent a bunch of time doing merge recursively and when I got to the bottom of the page I noticed it said to do it iteratively so that was a bit annoying.”

Feedback

We hear you…

“I like the stanford libraries but it would also be nice to see how coding is done in outside settings” **Moving forward, we will :)**

“I liked LaIR but I wish we didn't have to fill in a form to talk to one of the SLs.” **Come to office hours if you like a more relaxed setting!**

Feedback

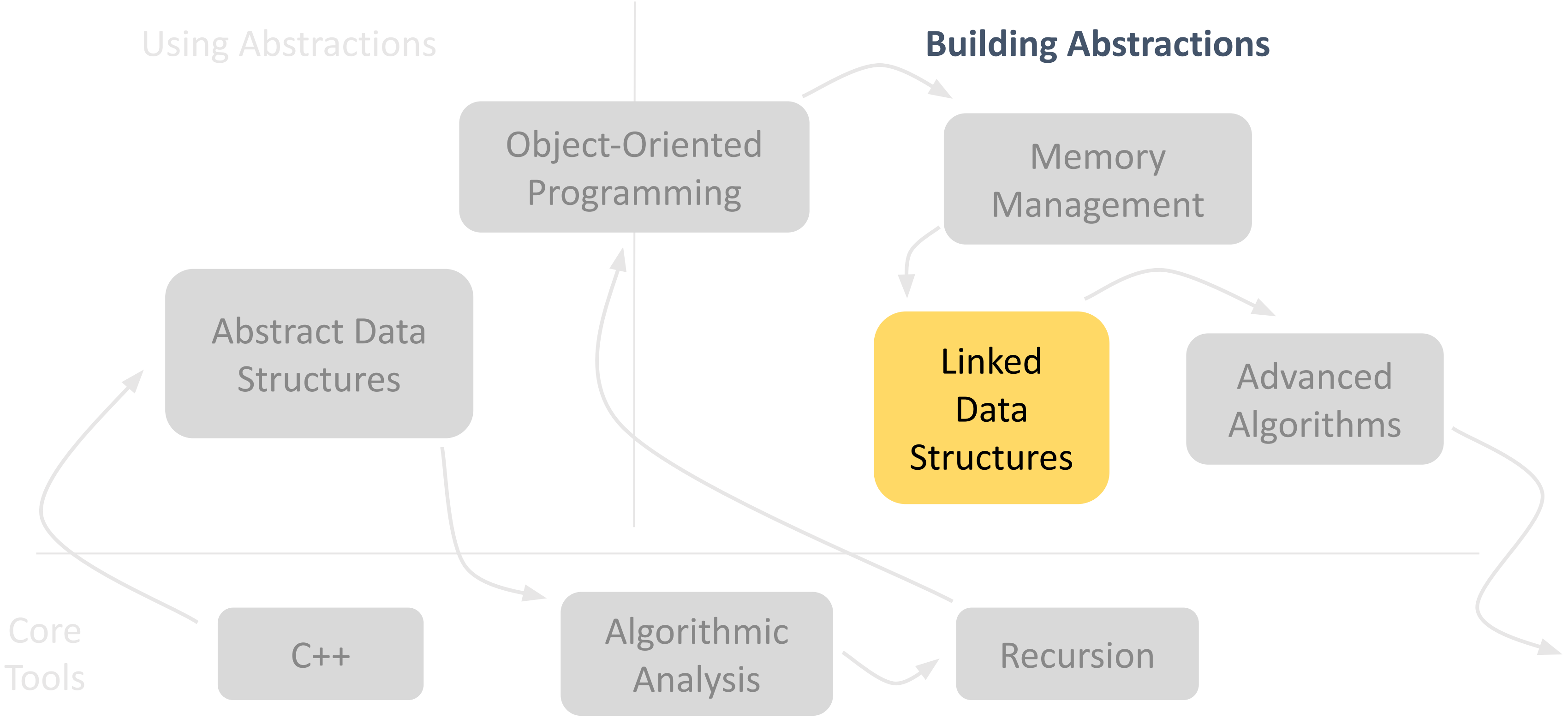
Anything else you would like us to know:

“It’s a char (as in charcoal) not a car. An array of cars is a parking lot, an array of chats is a string” **Controversial!**

“I am really considering CS for a major but I do not know what it would entail in the next few years.” **Come chat with us :)**

“I'm honestly not very fond of recursion, however I was able to appreciate the elegance of some solutions.” **Respect!**

Roadmap


Using Abstractions
Building Abstractions
Object-Oriented Programming
Memory Management
Abstract Data Structures
Linked Data Structures
Advanced Algorithms
Core Tools
C++
Algorithmic Analysis
Recursion

Recap: Linked Lists

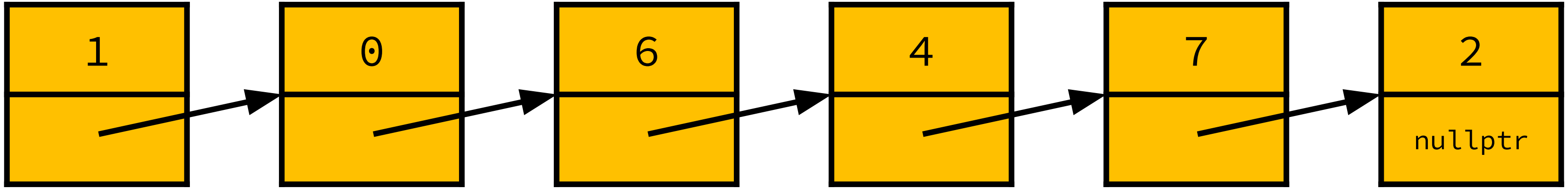
Frustrations with Arrays

- Not easily resizable
- Not efficient to insert elements at the beginning

1	0	6	4	7	2
0	1	2	3	4	5

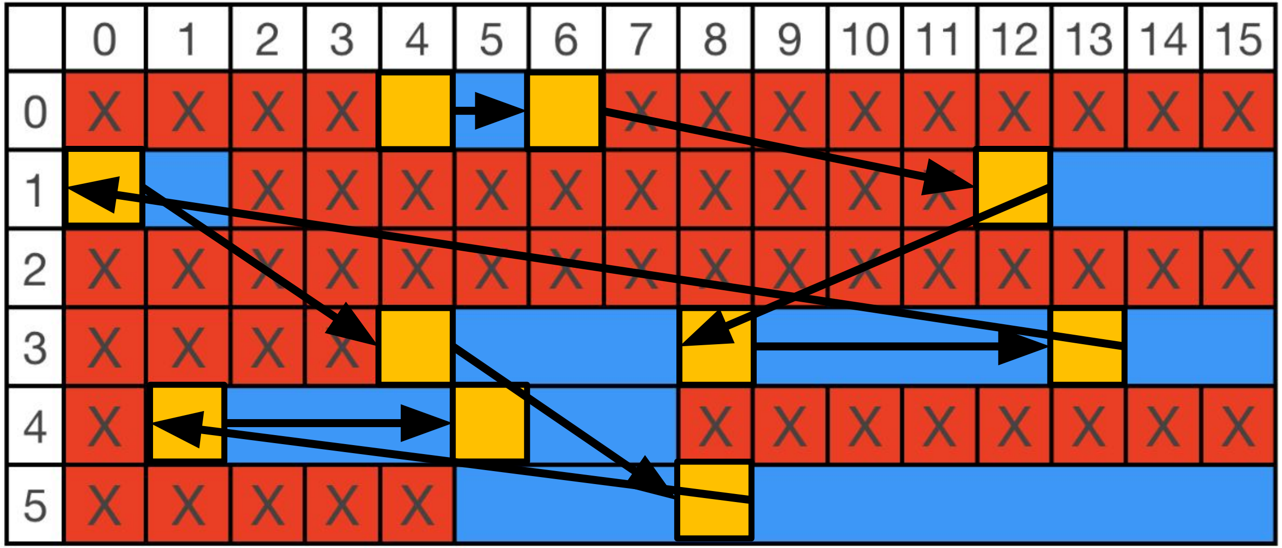
Benefits of Linked Lists

- Easily resizable
- Efficient to insert elements at the beginning



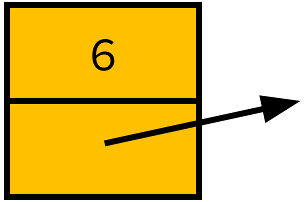
What are Linked Lists?

- A way we can use pointers to organize non-contiguous memory on the heap



Linked Lists, Structurally

- A linked list is a chain of nodes
- Each node is a struct that contains:
  - A piece of data (like an int, or string)
  - A pointer to the next node



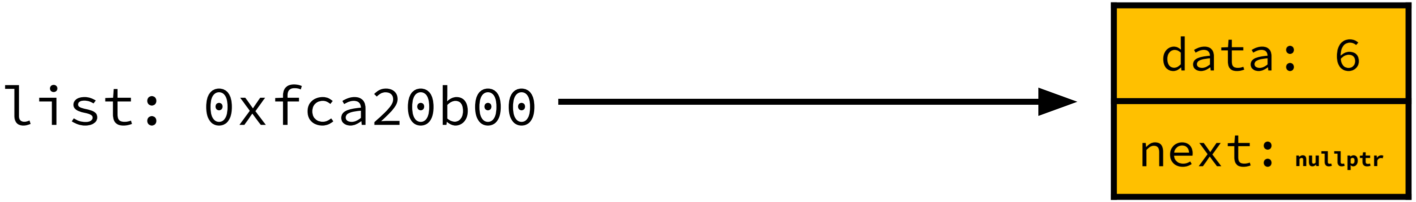

```
struct Node {
   int data;
   Node* next;
};
```

Creating a Linked List

- Create a new Node on the heap and store a pointer to it

```
Node* list = new Node;
list->data = 6;
list->next = nullptr;
```

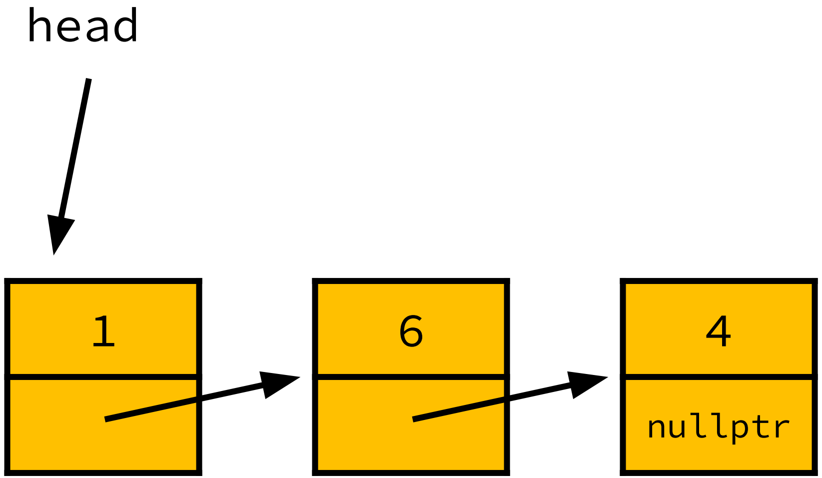
Dereference AND access the field for struct pointers using ->



Lives at `0xfca20b00` on the heap

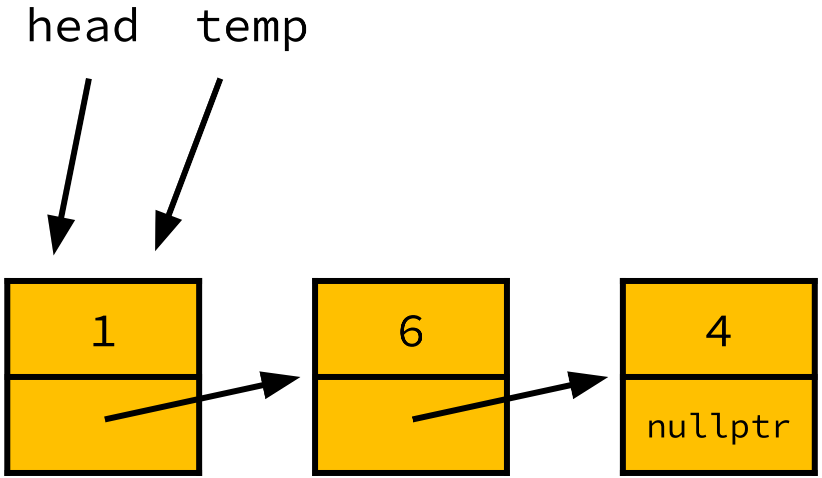
Review: Free Linked List

```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```



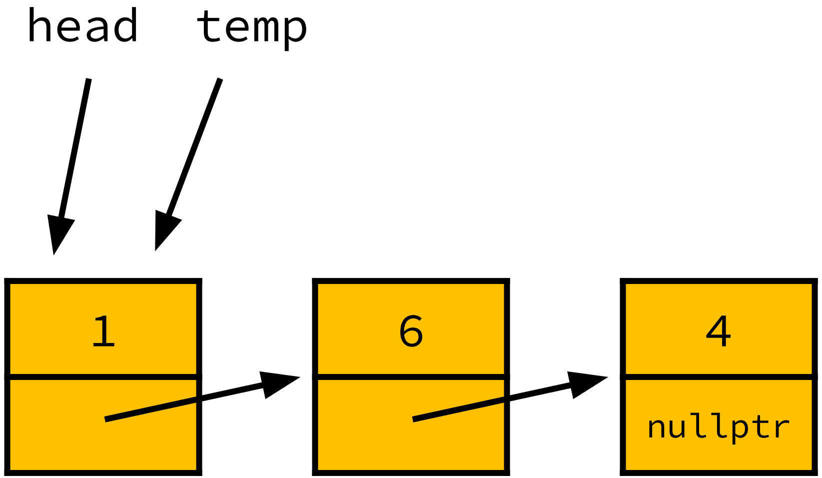
Review: Free Linked List

```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```



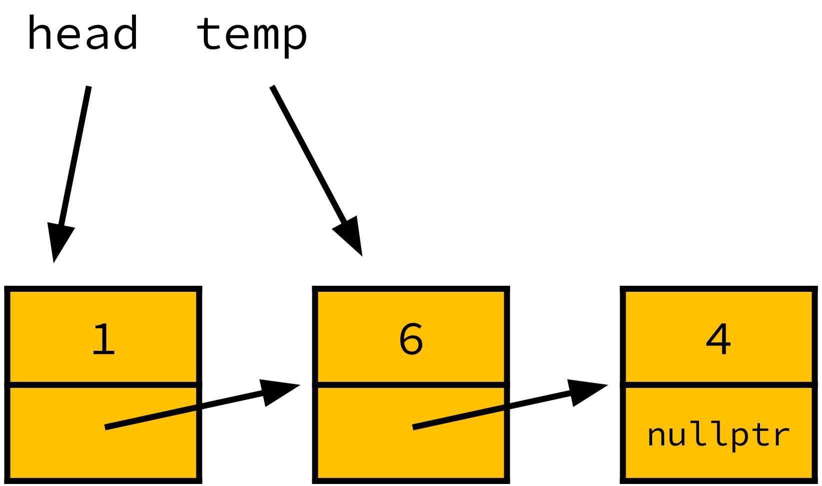
Review: Free Linked List

```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```



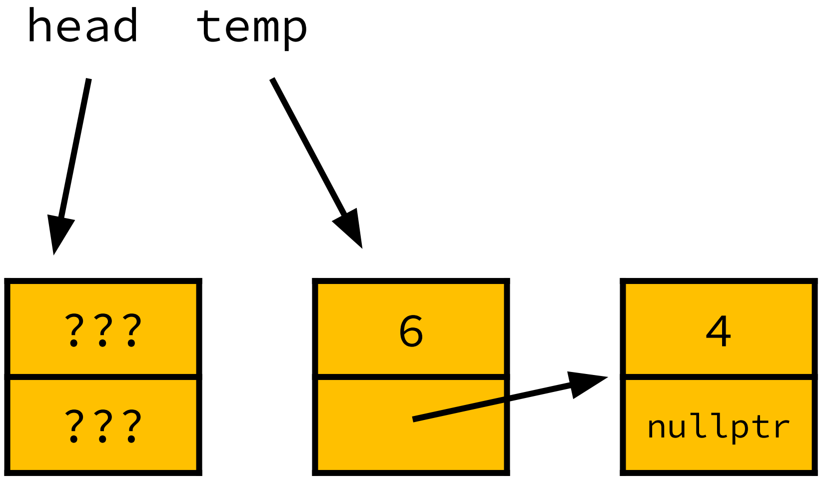
Review: Free Linked List

```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```



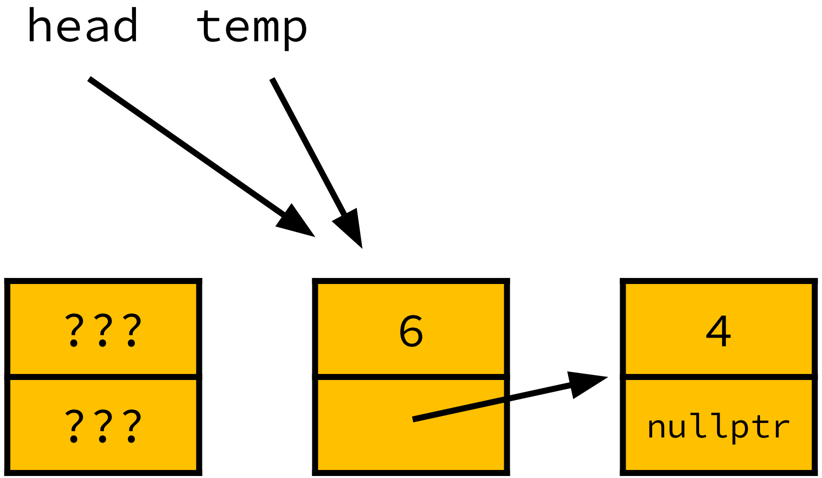
Review: Free Linked List

```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```



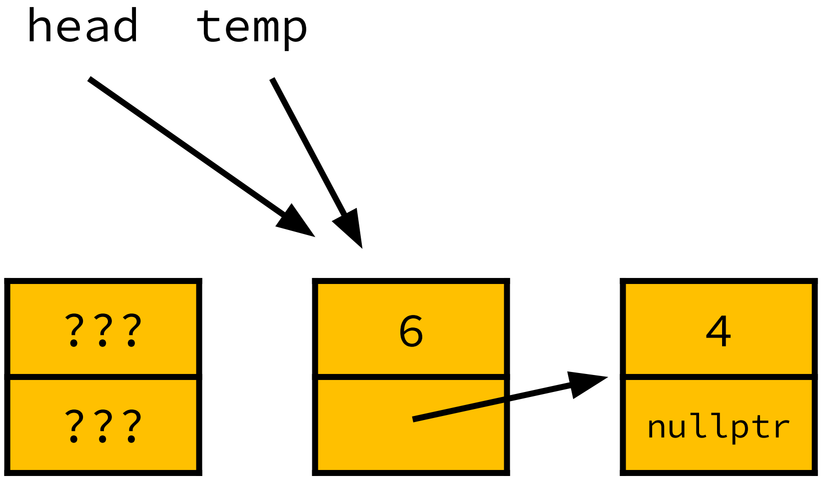
Review: Free Linked List

```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```



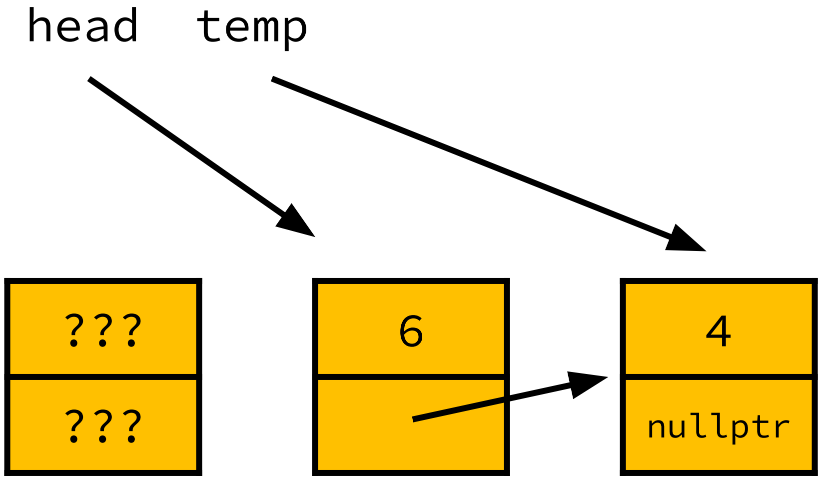
Review: Free Linked List

```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```



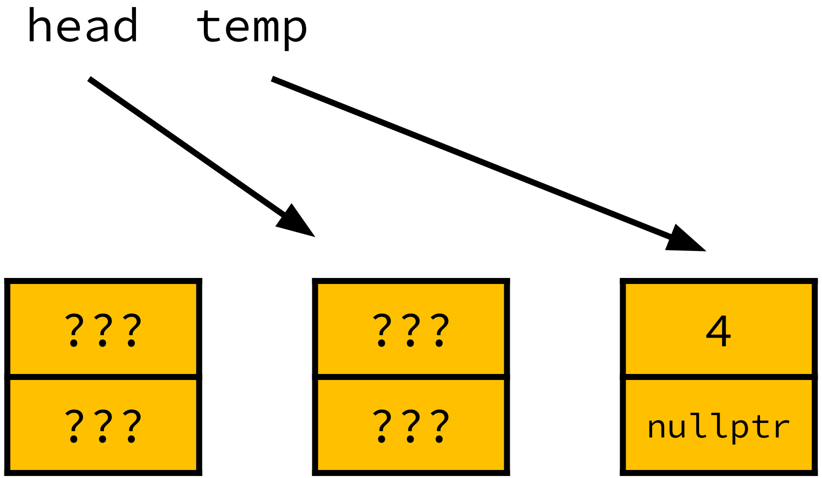
Review: Free Linked List

```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```



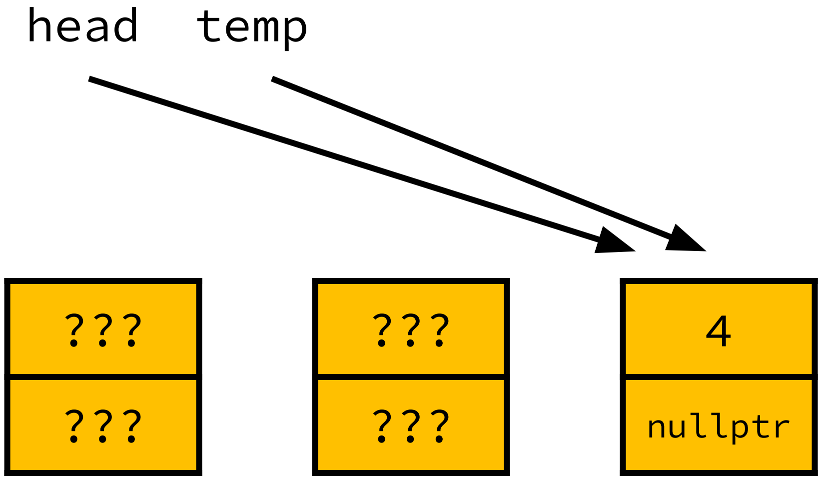
Review: Free Linked List

```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```



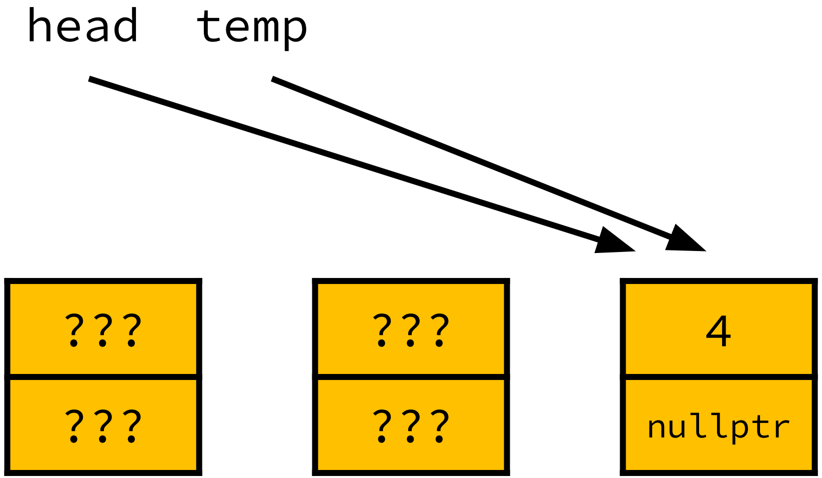
Review: Free Linked List

```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```



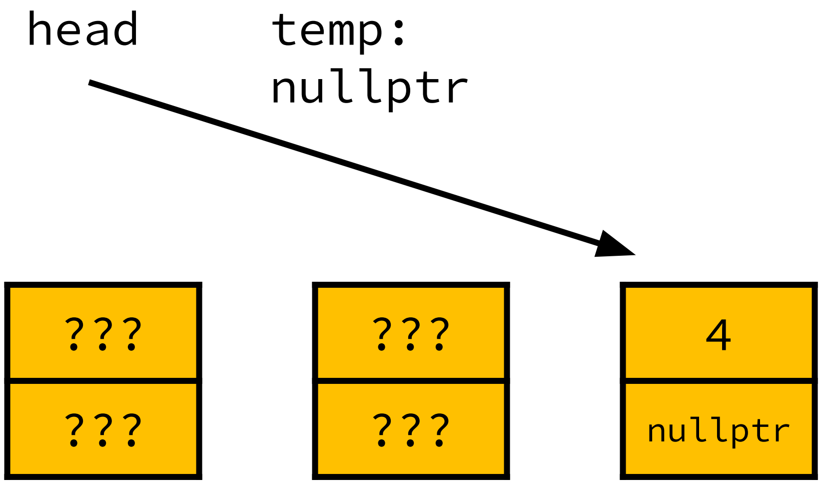
Review: Free Linked List

```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```



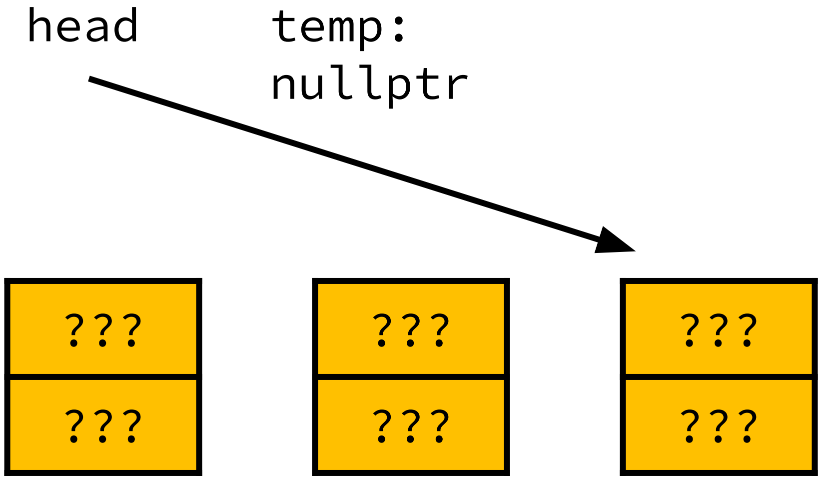
Review: Free Linked List

```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```



Review: Free Linked List

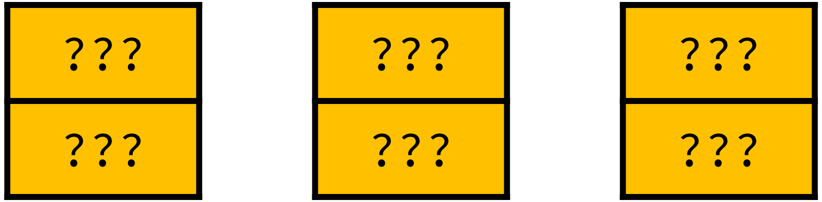
```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```



Review: Free Linked List

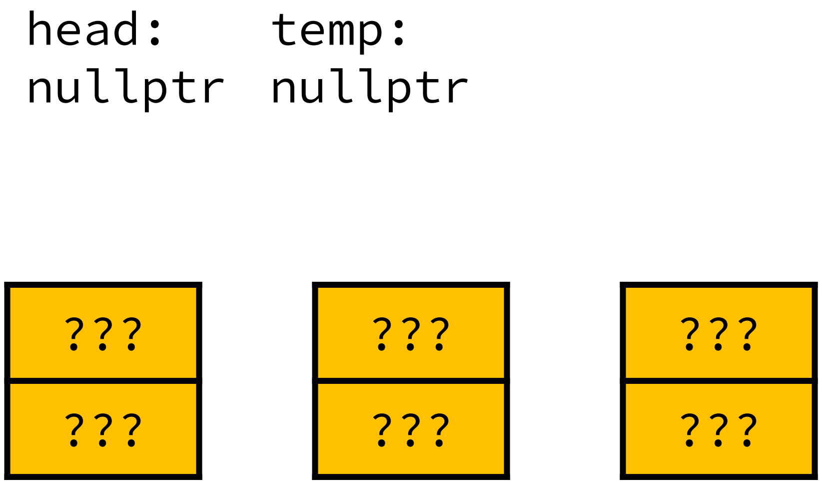
```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```

```
head:    temp:
nullptr  nullptr
```



Review: Free Linked List

```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```

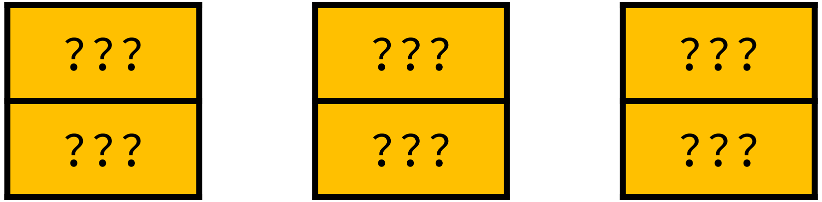


Review: Free Linked List

```
void freeList(Node* head) {
    Node* temp = head;
    while (head != nullptr) {
        temp = temp->next;
        delete head;
        head = temp;
    }
}
```

HAPPY TIMES 👍

head: nullptr   temp: nullptr



Linked Lists vs. Arrays

Linked Lists

- Chain of nodes, not contiguous in heap memory
- Access nodes starting at head, following the -> next pointer
- Good for implementing other data structures
- Has no member functions like `.size()` or `.add()`

Arrays

- Contiguous chunk of memory on the heap
- Access elements by index
- Same!
- Same!

Linked Lists and Recursion

Redefining Linked Lists

- Recall that the structure of a linked list Node is recursive:

```
struct Node {
    string data;
    Node* next;
};
```

Redefining Linked Lists

- Recall that the structure of a linked list `Node` is recursive:

```
struct Node {
    string data;
    Node* next;
};
```

On another level, we can define a linked list recursively…

Redefining Linked Lists

A **linked list** is either:

An empty list (`nullptr`)

Or a single node that points to another **linked list**

Redefining Linked Lists

A **linked list** is either:

An empty list (`nullptr`)

Or a single node that points to another **linked list**

We can define linked lists recursively, so can we implement linked list operations recursively?

Redefining Linked List Traversal

Last time:

```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

Redefining Linked List Traversal

Last time:

```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

Recursive approach:

```
void printListRec(Node* list) {
    // Base case
    // Recursive case
}
```

Redefining Linked List Traversal

Last time:

```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

Recursive approach:

```
void printListRec(Node* list) {
    // Base case
    if (list == nullptr) {
        return;
    }
    // Recursive case
}
```

Redefining Linked List Traversal

Last time:

```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}
```

Recursive approach:

```
void printListRec(Node* list) {
    // Base case
    if (list == nullptr) {
        return;
    }
    // Recursive case
    cout << list->data << endl;
    printListRec(list->next);
}
```

Pitfalls of Recursive List Traversal

- This recursive solution looks pretty elegant…

Pitfalls of Recursive List Traversal

- This recursive solution looks pretty elegant…
- However, note that the recursive solution generates one recursive call for every element in the list - a linked list with n elements would require n stack frames

Pitfalls of Recursive List Traversal

- This recursive solution looks pretty elegant…
- However, note that the recursive solution generates one recursive call for every element in the list - a linked list with n elements would require n stack frames
- For most computers, the stack frame limit is somewhere in the range of 16-64K - we can't traverse lists with more than 64K elements recursively!

Pitfalls of Recursive List Traversal

- This recursive solution looks pretty elegant…
- Howev … recursive call fo … ents would
- For mo … in the range of 16-64K - we can't traverse lists with more than 64K elements recursively!

On Assignment 5, avoid doing list traversals recursively! Today, we'll see that which operations entail some kind of traversal.

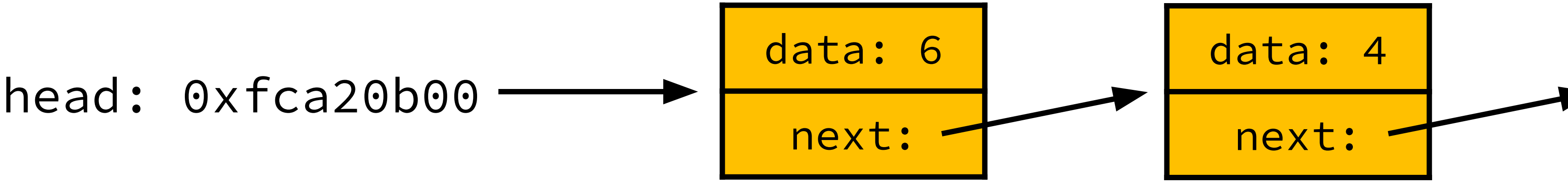
Big-O of Linked List Operations

Linked List Operations

- Prepend
- Append
- Insert
- Delete
- Traverse

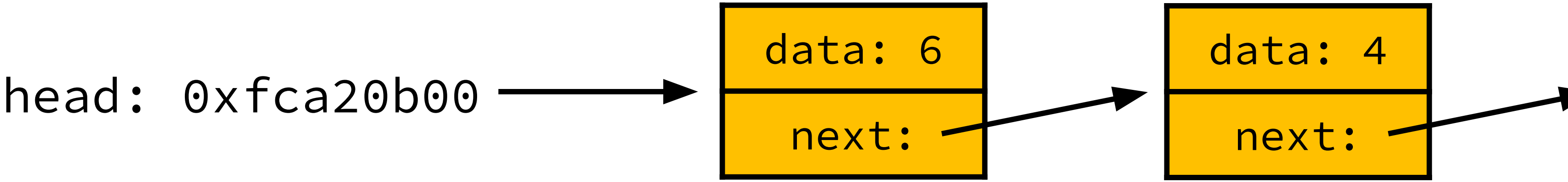
Linked List Prepend

- Create a node, and make this the new head of the list
- `O(1)` - no relation to the length of our list n



Linked List Prepend

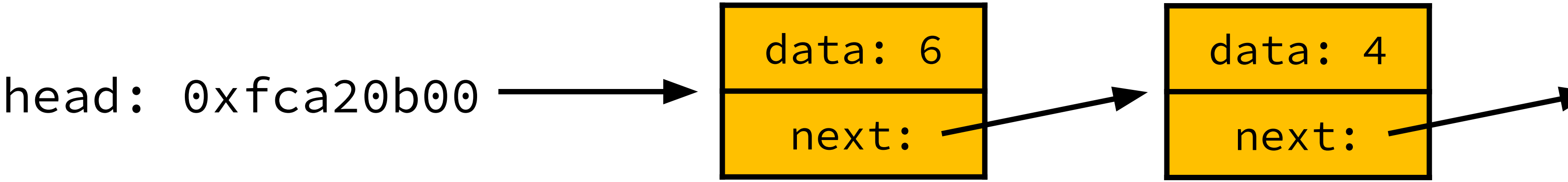
- Create a node, and make this the new head of the list
- `O(1)` - no relation to the length of our list n



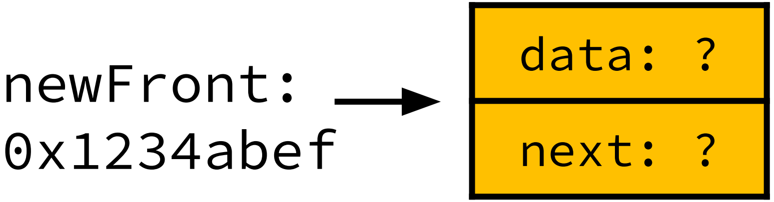

```
Node* newFront = new Node;
newFront->data = 1;
newFront->next = head;
head = newFront;
```

Linked List Prepend

- Create a node, and make this the new head of the list
- `O(1)` - no relation to the length of our list n

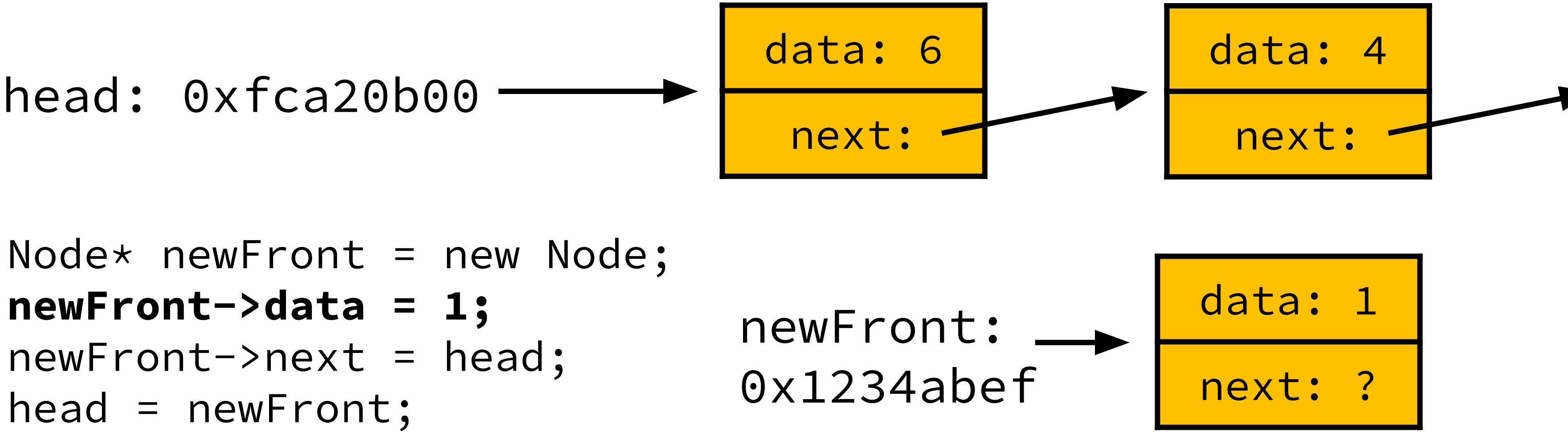



```
Node* newFront = new Node;
newFront->data = 1;
newFront->next = head;
head = newFront;
```



Linked List Prepend

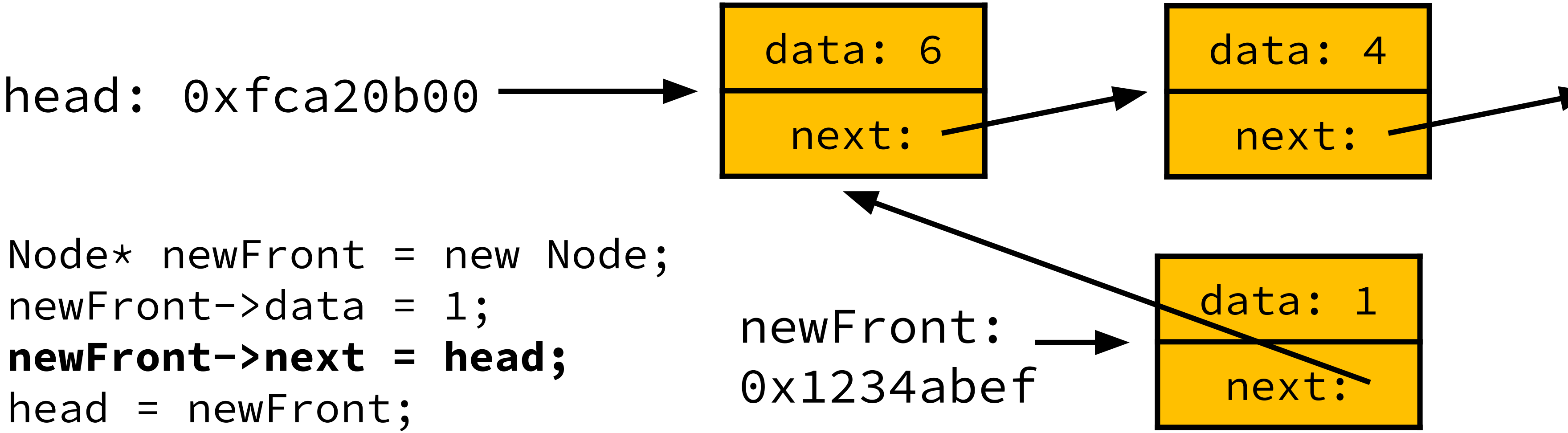
- Create a node, and make this the new head of the list
- `O(1)` - no relation to the length of our list n




```
Node* newFront = new Node;
newFront->data = 1;
newFront->next = head;
head = newFront;
```

Linked List Prepend

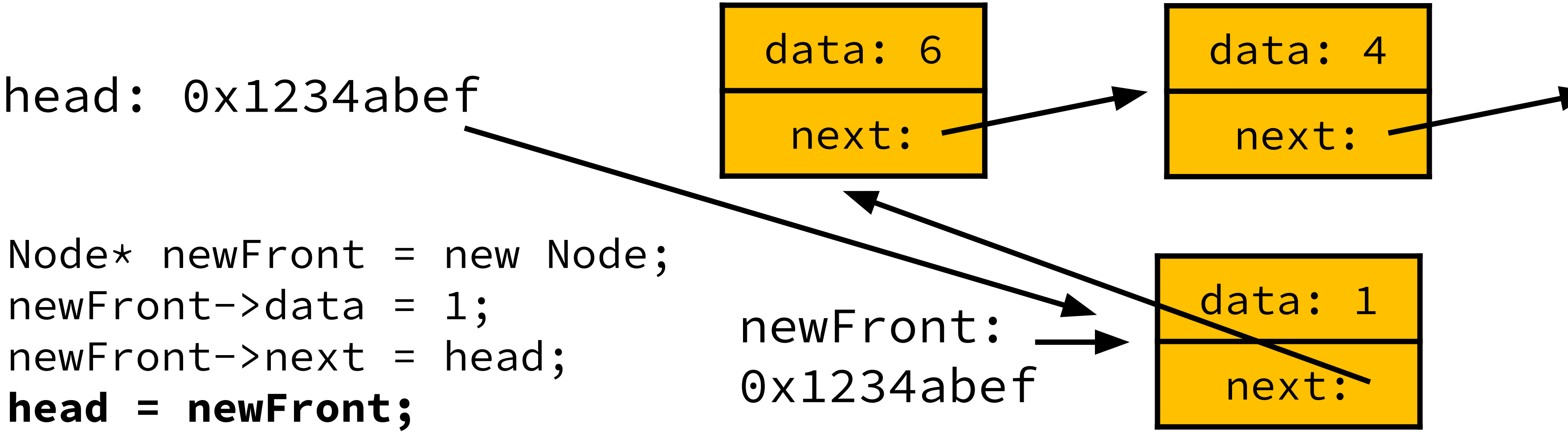
- Create a node, and make this the new head of the list
- `O(1)` - no relation to the length of our list n




```
Node* newFront = new Node;
newFront->data = 1;
newFront->next = head;
head = newFront;
```

Linked List Prepend

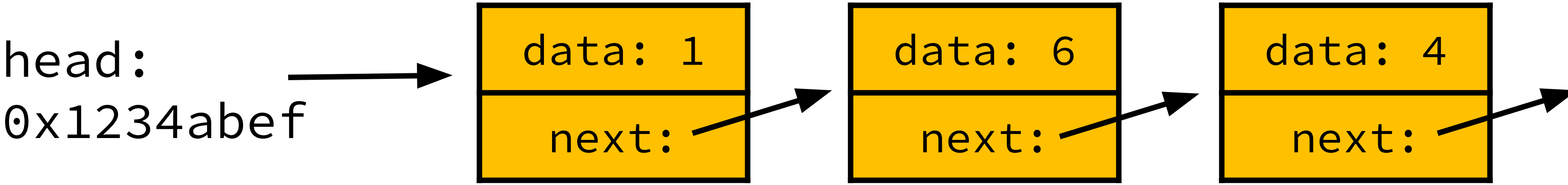
- Create a node, and make this the new head of the list
- `O(1)` - no relation to the length of our list n




```
Node* newFront = new Node;
newFront->data = 1;
newFront->next = head;
head = newFront;
```

Linked List Prepend

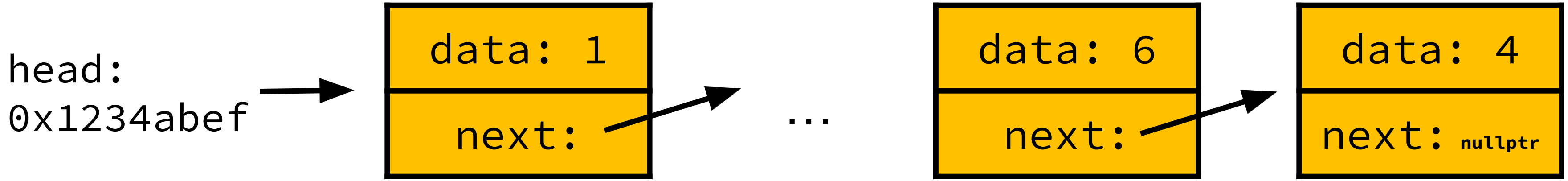
- Create a node, and make this the new head of the list
- `O(1)` - no relation to the length of our list n




```
Node* newFront = new Node;
newFront->data = 1;
newFront->next = head;
head = newFront;
```

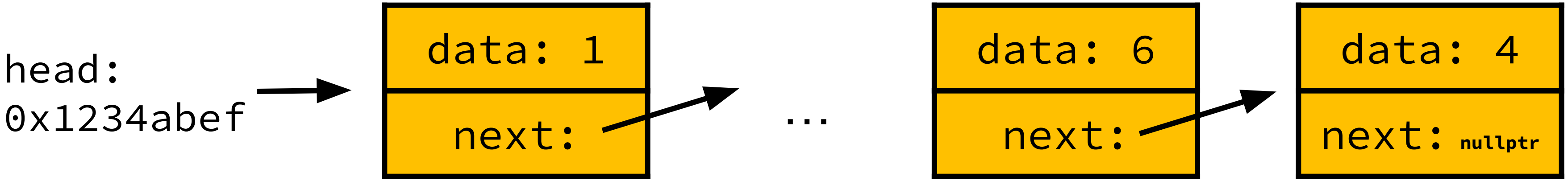
Linked List Append

- Traverse to the end of our list, create and link in new node
- `O(n)` - we have to visit n other nodes before reaching the end



Linked List Append

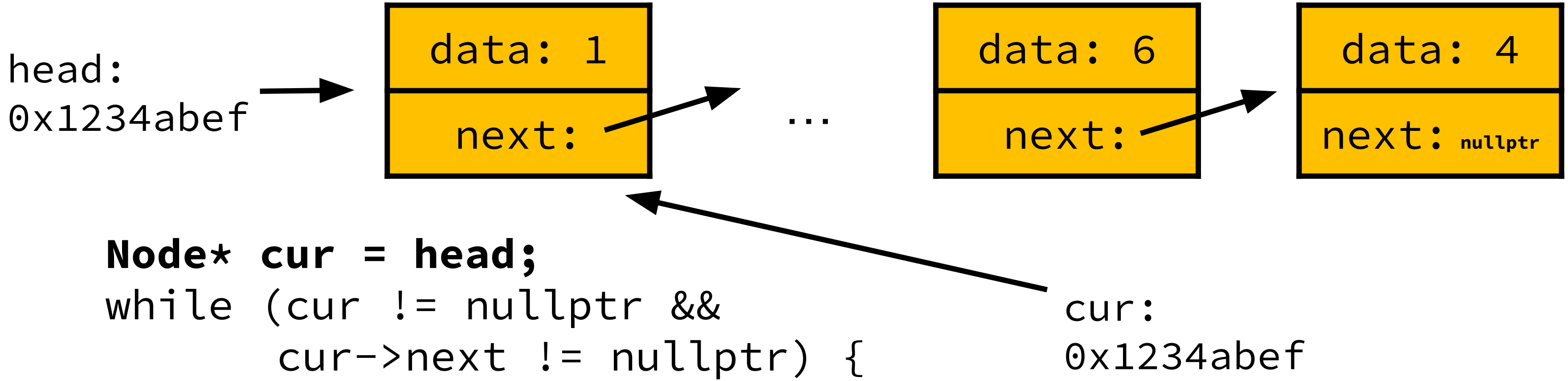
- **Traverse to the end of our list**, create and link in new node
- `O(n)` - we have to visit n other nodes before reaching the end




```
Node* cur = head;
while (cur != nullptr &&
        cur->next != nullptr) {
    cur = cur->next;
}
```

Linked List Append

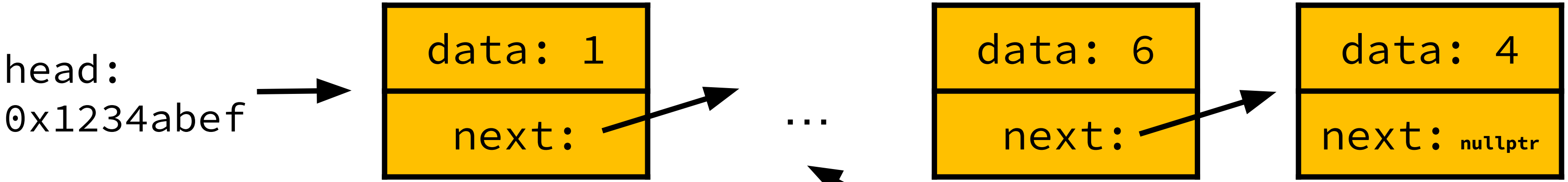
- **Traverse to the end of our list**, create and link in new node
- O(n) - we have to visit n other nodes before reaching the end




```
Node* cur = head;
while (cur != nullptr &&
        cur->next != nullptr) {
    cur = cur->next;
}
```

Linked List Append

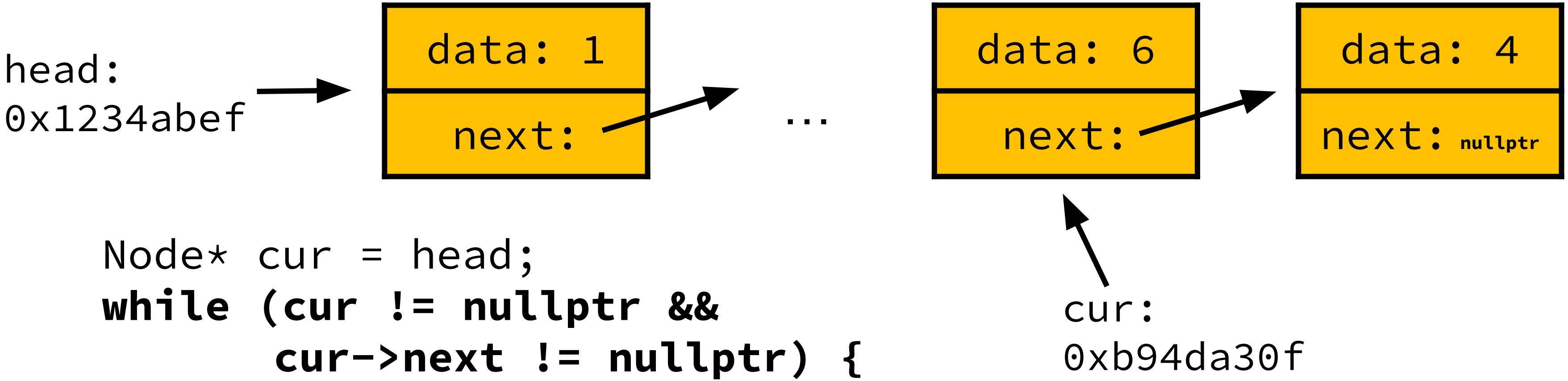
- **Traverse to the end of our list**, create and link in new node
- `O(n)` - we have to visit n other nodes before reaching the end




```
Node* cur = head;
while (cur != nullptr &&
        cur->next != nullptr) {
    cur = cur->next;
}
```

Linked List Append

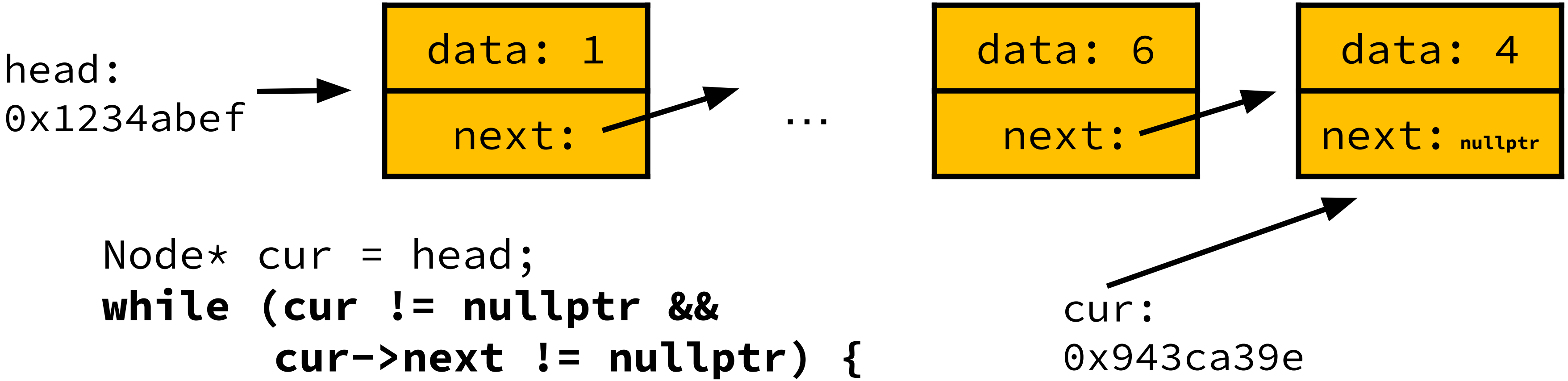
- **Traverse to the end of our list**, create and link in new node
- `O(n)` - we have to visit n other nodes before reaching the end




```
Node* cur = head;
while (cur != nullptr &&
        cur->next != nullptr) {
    cur = cur->next;
}
```

Linked List Append

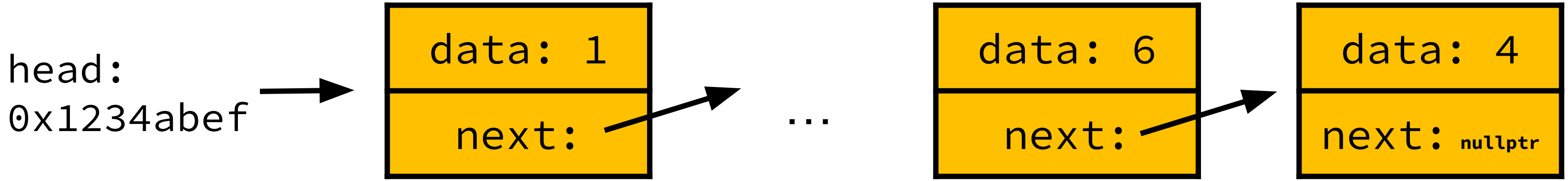
- **Traverse to the end of our list**, create and link in new node
- `O(n)` - we have to visit n other nodes before reaching the end

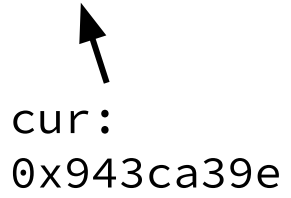



```
Node* cur = head;
while (cur != nullptr &&
        cur->next != nullptr) {
    cur = cur->next;
}
```

Linked List Append

- Traverse to the end of our list, **create and link in new node**
- `O(n)` - we have to visit n other nodes before reaching the end

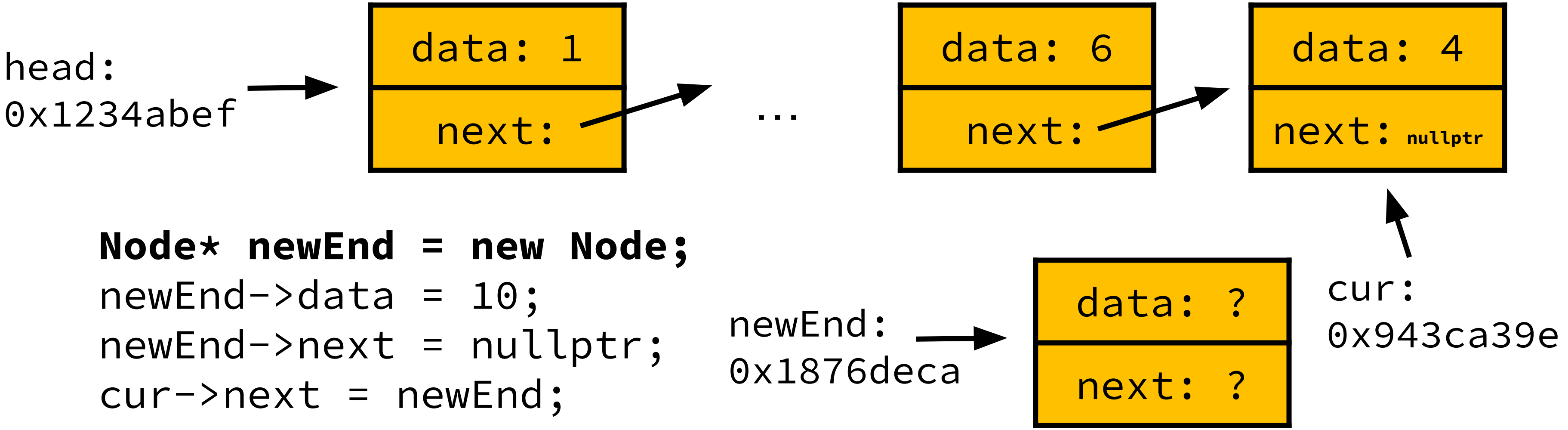





```
Node* newEnd = new Node;
newEnd->data = 10;
newEnd->next = nullptr;
cur->next = newEnd;
```

Linked List Append

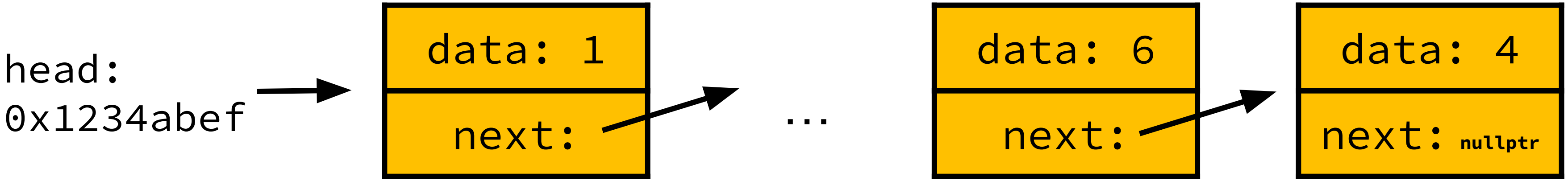
- Traverse to the end of our list, **create and link in new node**
- `O(n)` - we have to visit n other nodes before reaching the end



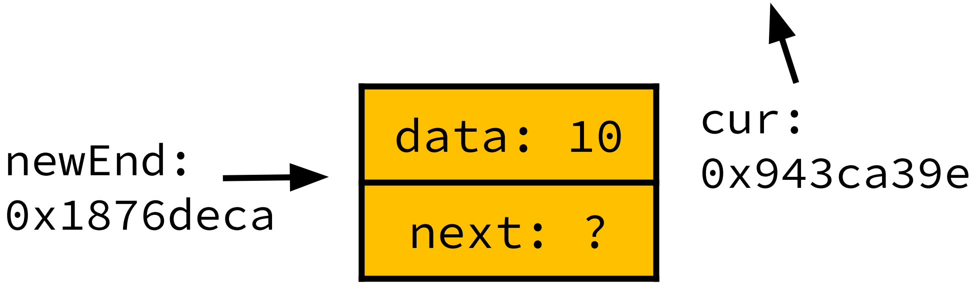

```
Node* newEnd = new Node;
newEnd->data = 10;
newEnd->next = nullptr;
cur->next = newEnd;
```

Linked List Append

- Traverse to the end of our list, **create and link in new node**
- O(n) - we have to visit n other nodes before reaching the end

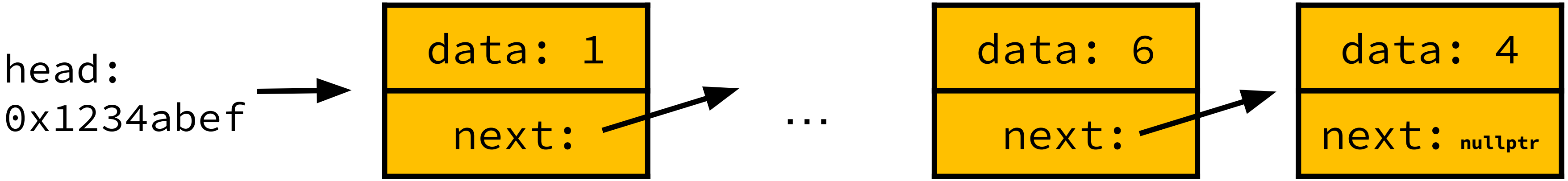



```
Node* newEnd = new Node;
newEnd->data = 10;
newEnd->next = nullptr;
cur->next = newEnd;
```

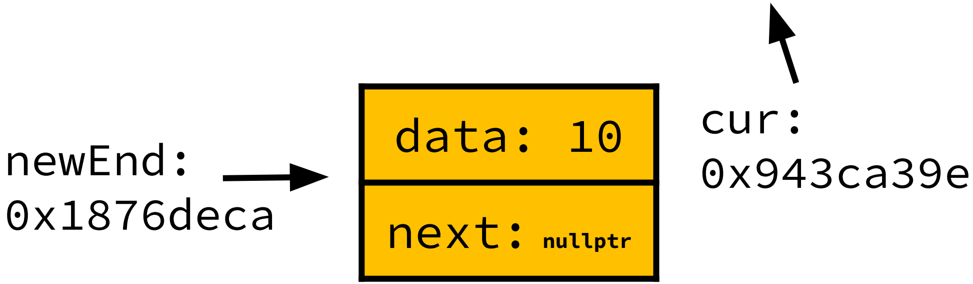


Linked List Append

- Traverse to the end of our list, **create and link in new node**
- `O(n)` - we have to visit n other nodes before reaching the end

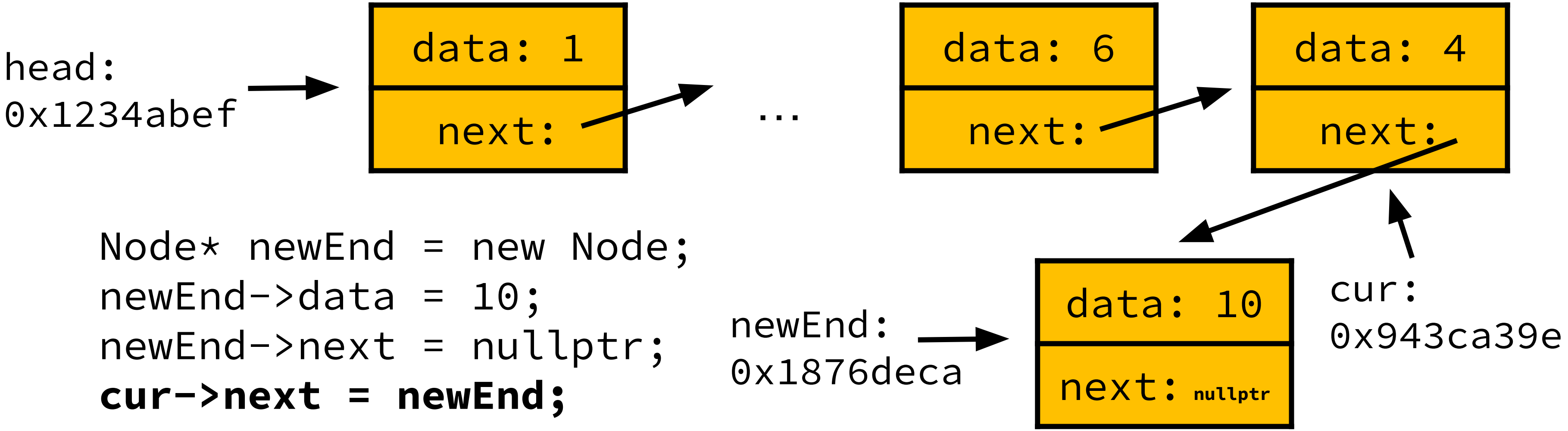



```
Node* newEnd = new Node;
newEnd->data = 10;
newEnd->next = nullptr;
cur->next = newEnd;
```



Linked List Append

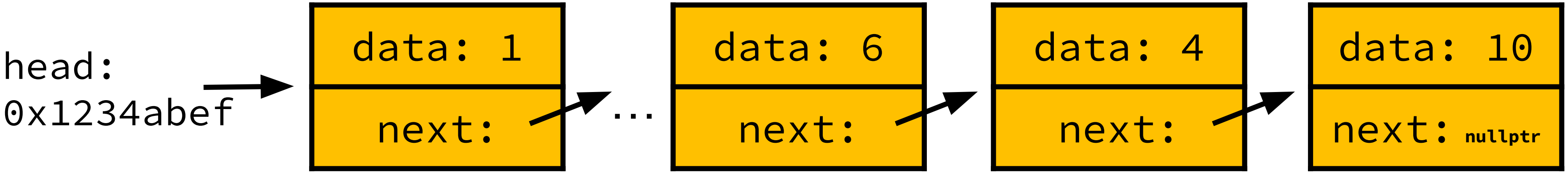
- Traverse to the end of our list, **create and link in new node**
- `O(n)` - we have to visit n other nodes before reaching the end




```
Node* newEnd = new Node;
newEnd->data = 10;
newEnd->next = nullptr;
cur->next = newEnd;
```

Linked List Append

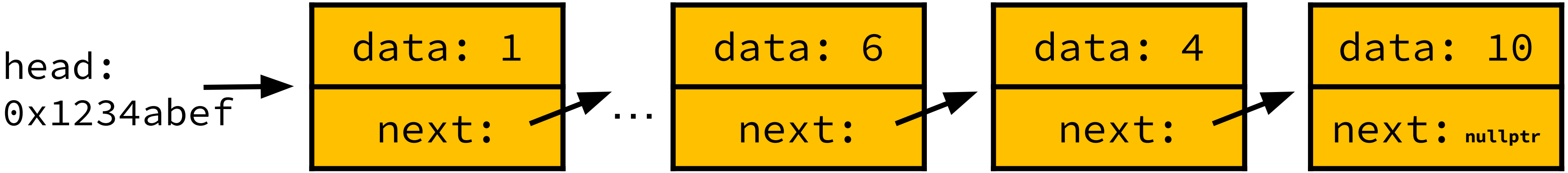
- Traverse to the end of our list, **create and link in new node**
- `O(n)` - we have to visit n other nodes before reaching the end




```
Node* newEnd = new Node;
newEnd->data = 10;
newEnd->next = nullptr;
cur->next = newEnd;
```

Linked List Append

- Traverse to the end of our list, create and link in new node
- `O(n)` - we have to visit n other nodes before reaching the end

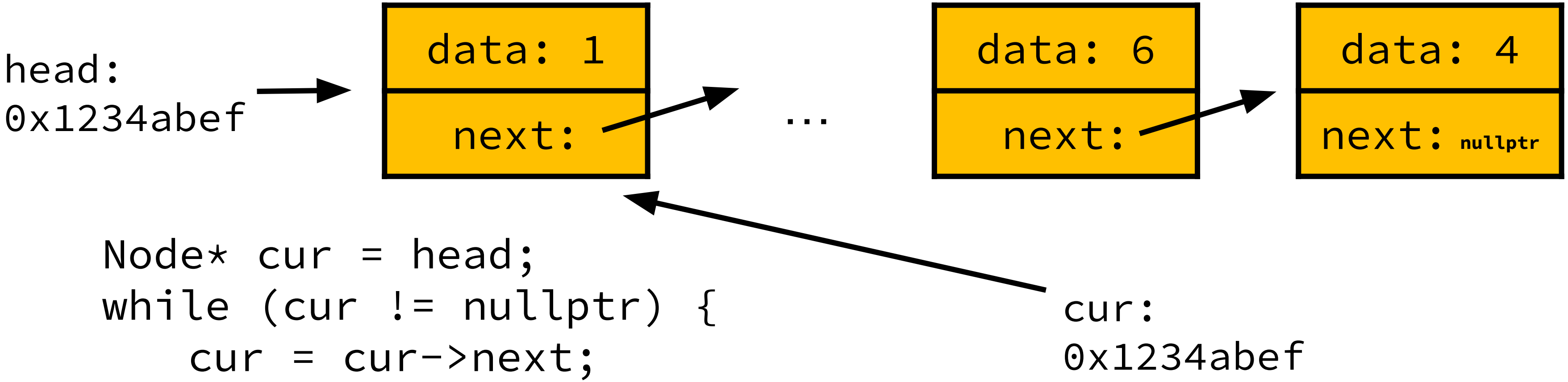



```
Node* cur = head;
while (cur != nullptr &&
        cur->next != nullptr) {
    cur = cur->next;
}
```

👥 *Why did we have this condition in our traversal loop?*

Linked List Append

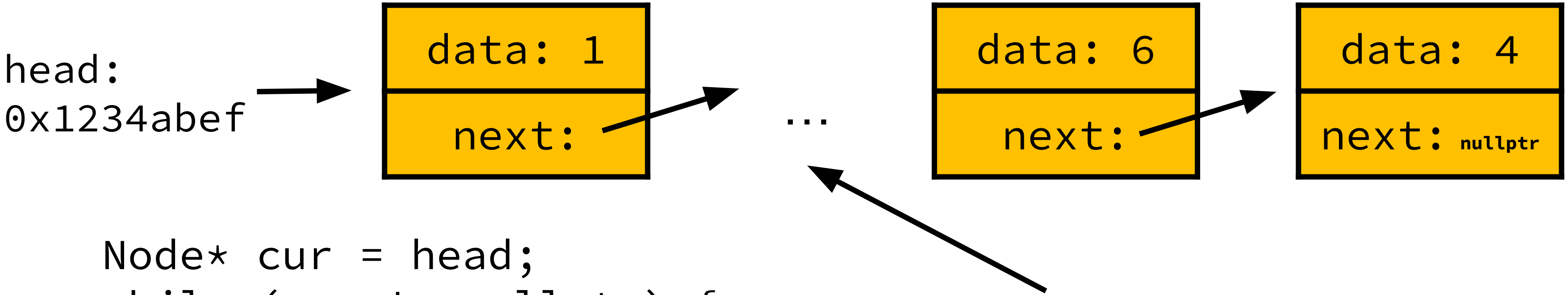
- **Traverse to the end of our list**, create and link in new node
- `O(n)` - we have to visit n other nodes before reaching the end




```
Node* cur = head;
while (cur != nullptr) {
    cur = cur->next;
}
```

Linked List Append

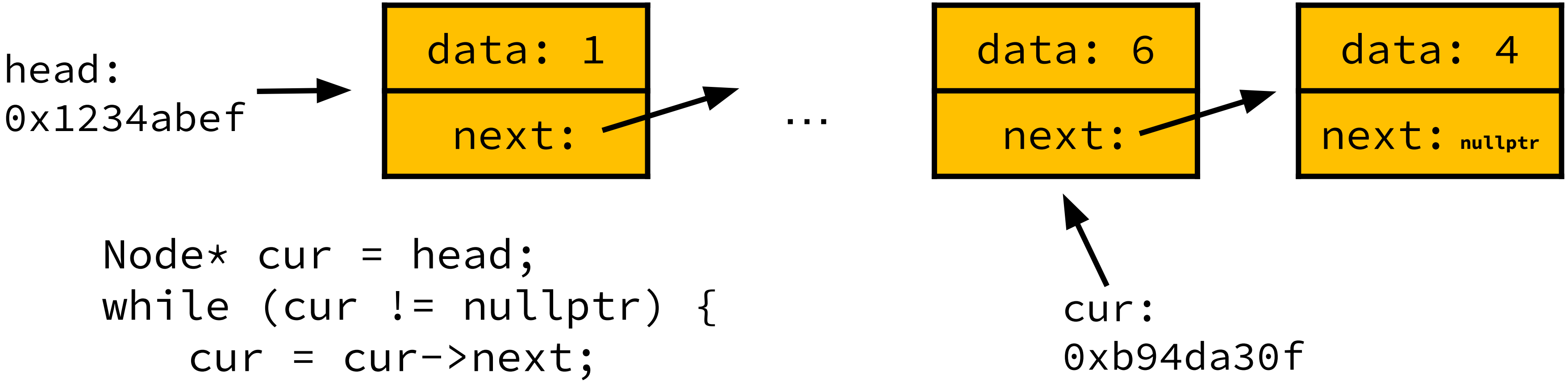
- **Traverse to the end of our list**, create and link in new node
- `O(n)` - we have to visit n other nodes before reaching the end




```
Node* cur = head;
while (cur != nullptr) {
    cur = cur->next;
}
```

Linked List Append

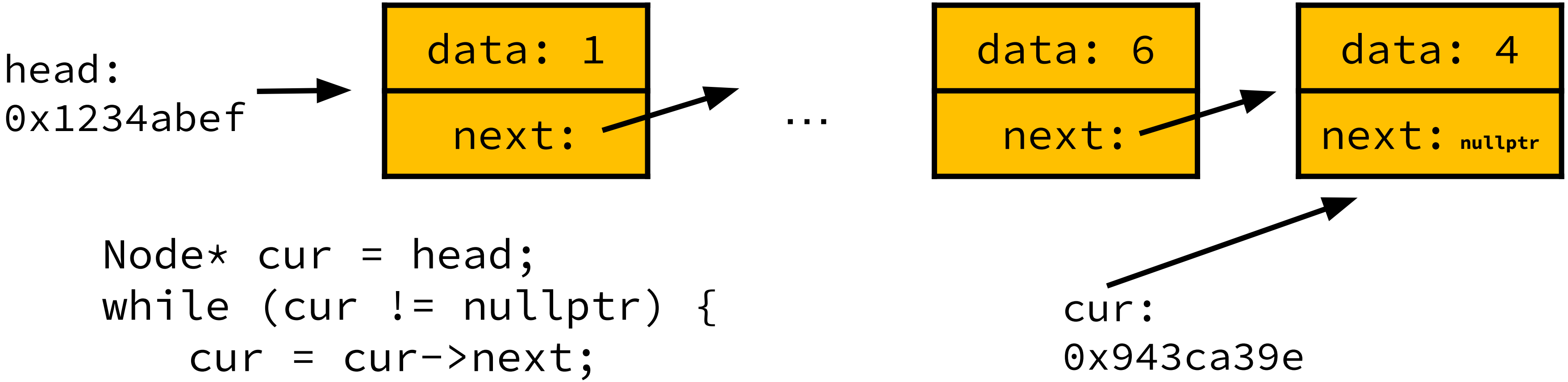
- **Traverse to the end of our list**, create and link in new node
- O(n) - we have to visit n other nodes before reaching the end




```
Node* cur = head;
while (cur != nullptr) {
    cur = cur->next;
}
```

Linked List Append

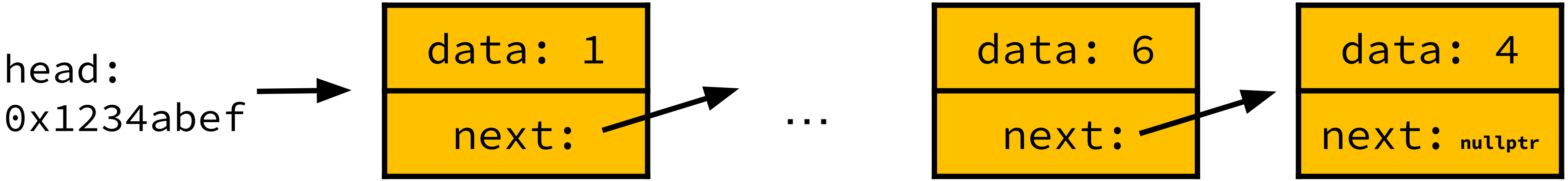
- **Traverse to the end of our list**, create and link in new node
- `O(n)` - we have to visit n other nodes before reaching the end




```
Node* cur = head;
while (cur != nullptr) {
    cur = cur->next;
}
```

Linked List Append

- **Traverse to the end of our list**, create and link in new node
- O(n) - we have to visit n other nodes before reaching the end

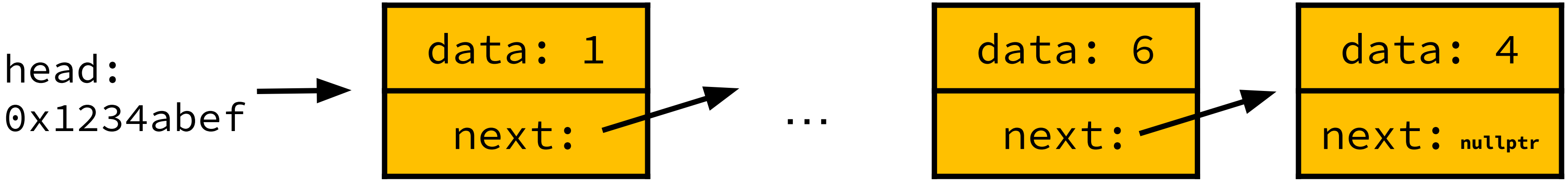



```
Node* cur = head;
while (cur != nullptr) {
    cur = cur->next;
}
```

cur:
nullptr

Linked List Append

- **Traverse to the end of our list**, create and link in new node
- `O(n)` - we have to visit n other nodes before reaching the end

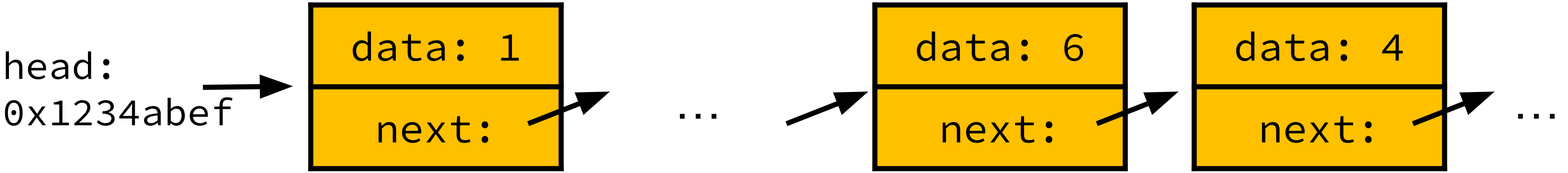



```
Node* cur = head;
while (cur != nullptr) {
    cur = cur->next;
}
```

To avoid "falling off" the end of our linked list!

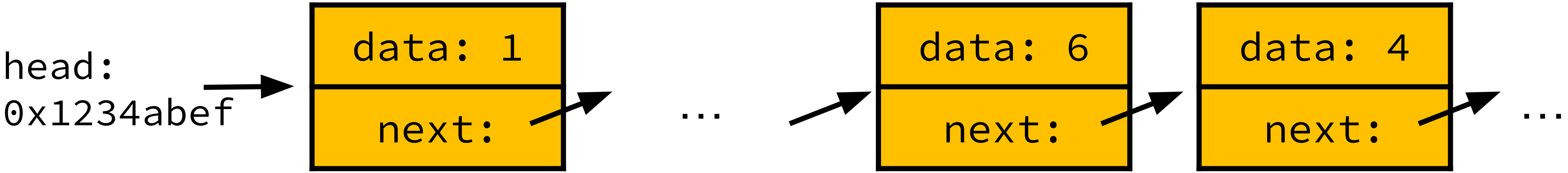
Linked List Insert

- Traverse to some location, create and link in new node
- `O(n)` - we have to visit `O(n)` other nodes before reaching location



Linked List Insert

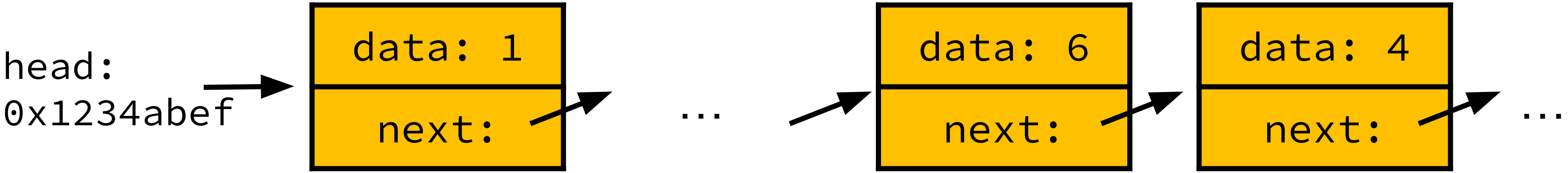
- Traverse to some location, create and link in new node
- `O(n)` - we have to visit `O(n)` other nodes before reaching location



Insert 5 after 6

Linked List Insert

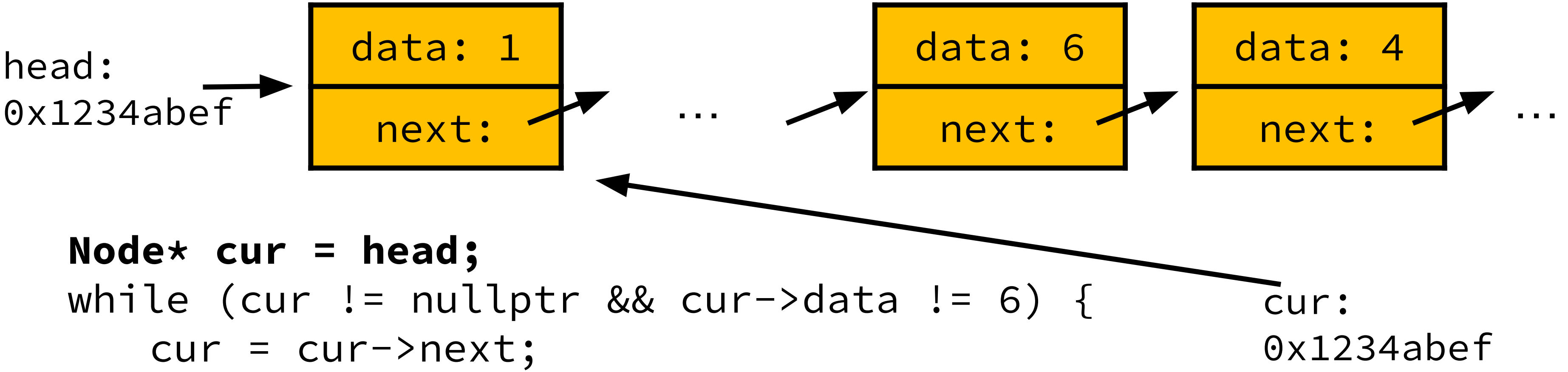
- **Traverse to some location**, create and link in new node
- `O(n)` - we have to visit `O(n)` other nodes before reaching location




```
Node* cur = head;
while (cur != nullptr && cur->data != 6) {
    cur = cur->next;
}
```

Linked List Insert

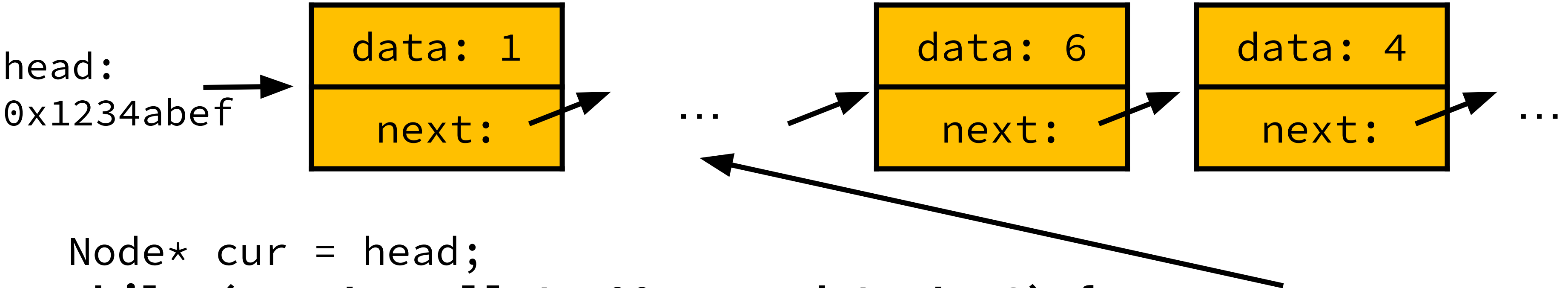
- **Traverse to some location**, create and link in new node
- `O(n)` - we have to visit `O(n)` other nodes before reaching location




```
Node* cur = head;
while (cur != nullptr && cur->data != 6) {
    cur = cur->next;
}
```

Linked List Insert

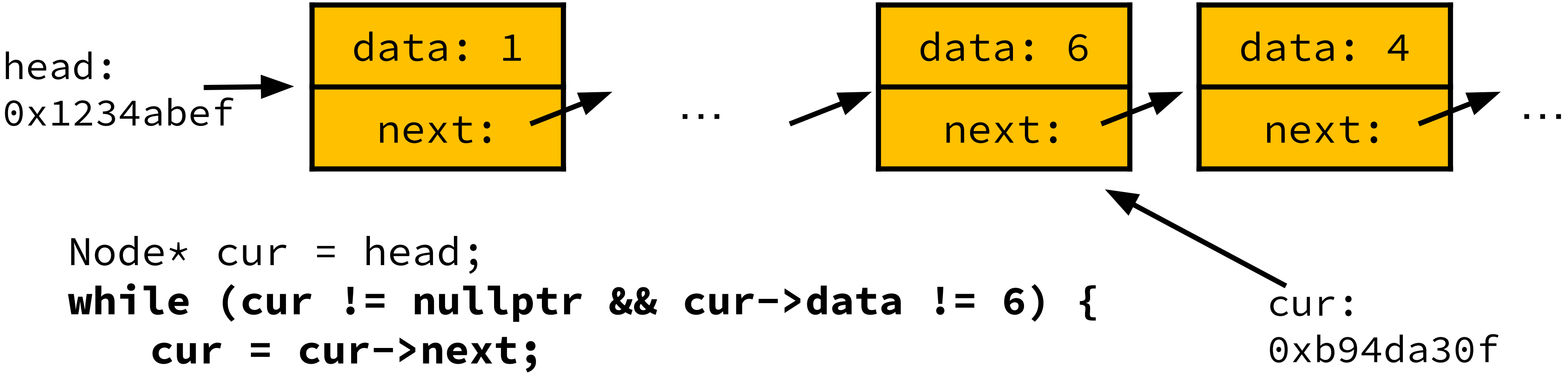
- **Traverse to some location**, create and link in new node
- `O(n)` - we have to visit `O(n)` other nodes before reaching location




```
Node* cur = head;
while (cur != nullptr && cur->data != 6) {
    cur = cur->next;
}
```

Linked List Insert

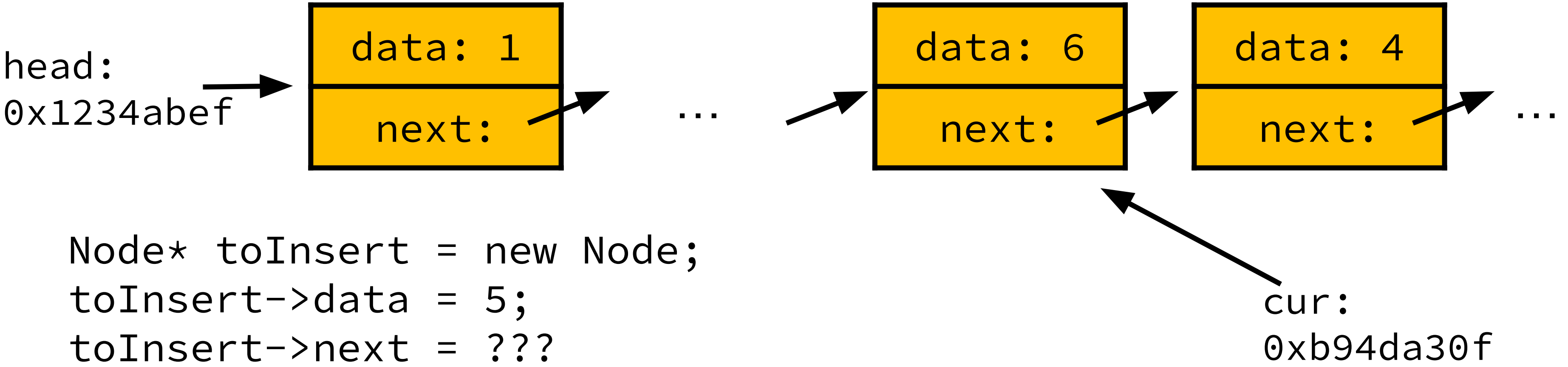
- **Traverse to some location**, create and link in new node
- `O(n)` - we have to visit `O(n)` other nodes before reaching location




```
Node* cur = head;
while (cur != nullptr && cur->data != 6) {
    cur = cur->next;
}
```

Linked List Insert

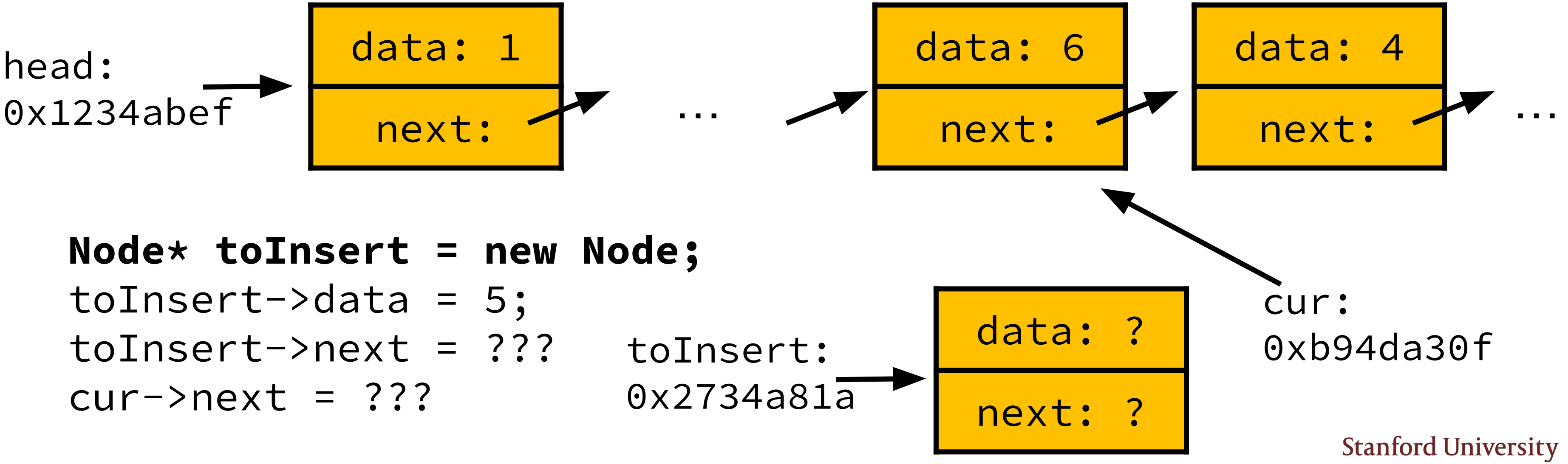
- Traverse to some location, **create and link in new node**
- `O(n)` - we have to visit `O(n)` other nodes before reaching location




```
Node* toInsert = new Node;
toInsert->data = 5;
toInsert->next = ???
cur->next = ???
```

Linked List Insert

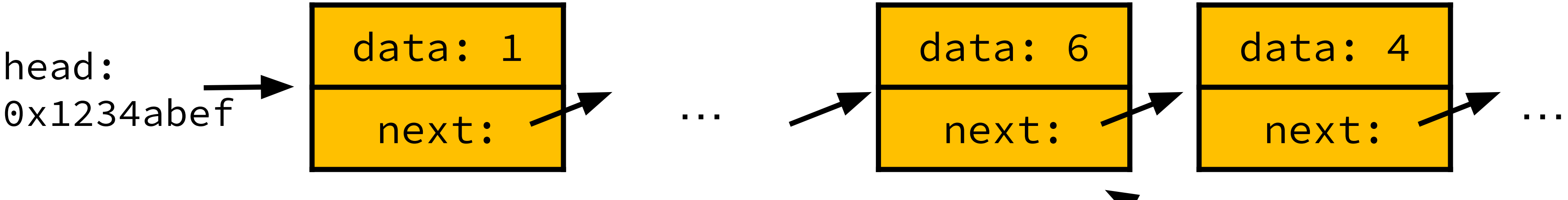
- Traverse to some location, **create and link in new node**
- `O(n)` - we have to visit `O(n)` other nodes before reaching location



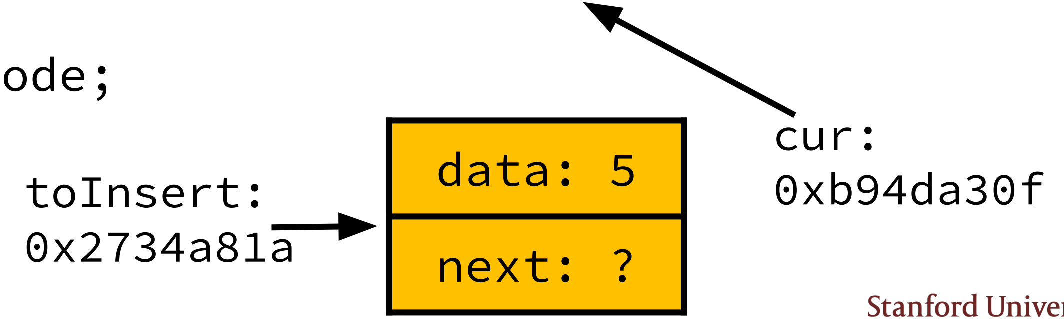

```
Node* toInsert = new Node;
toInsert->data = 5;
toInsert->next = ???
cur->next = ???
```

Linked List Insert

- Traverse to some location, **create and link in new node**
- `O(n)` - we have to visit `O(n)` other nodes before reaching location



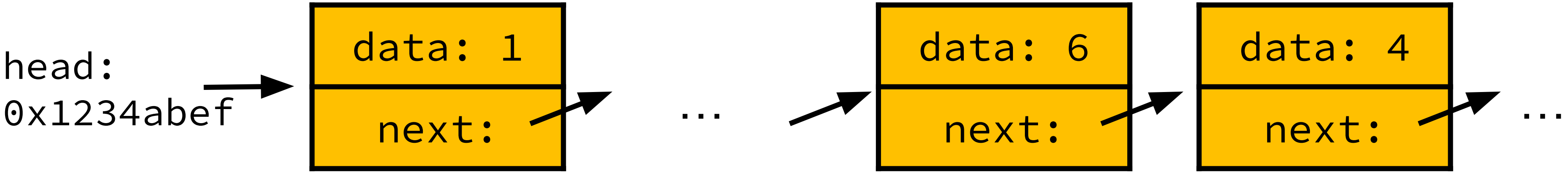

```
Node* toInsert = new Node;
toInsert->data = 5;
toInsert->next = ???
cur->next = ???
```



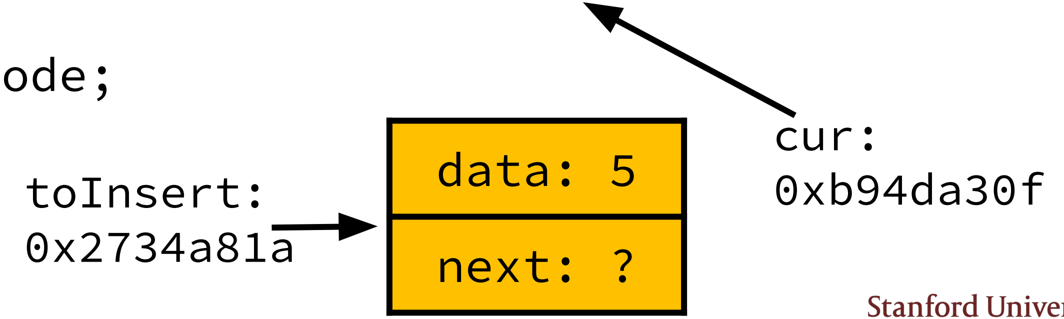
Linked List Insert

🤔 *How do we link in this new node?*

- Traverse to some location, **create and link in new node**
- `O(n)` - we have to visit `O(n)` other nodes before reaching location

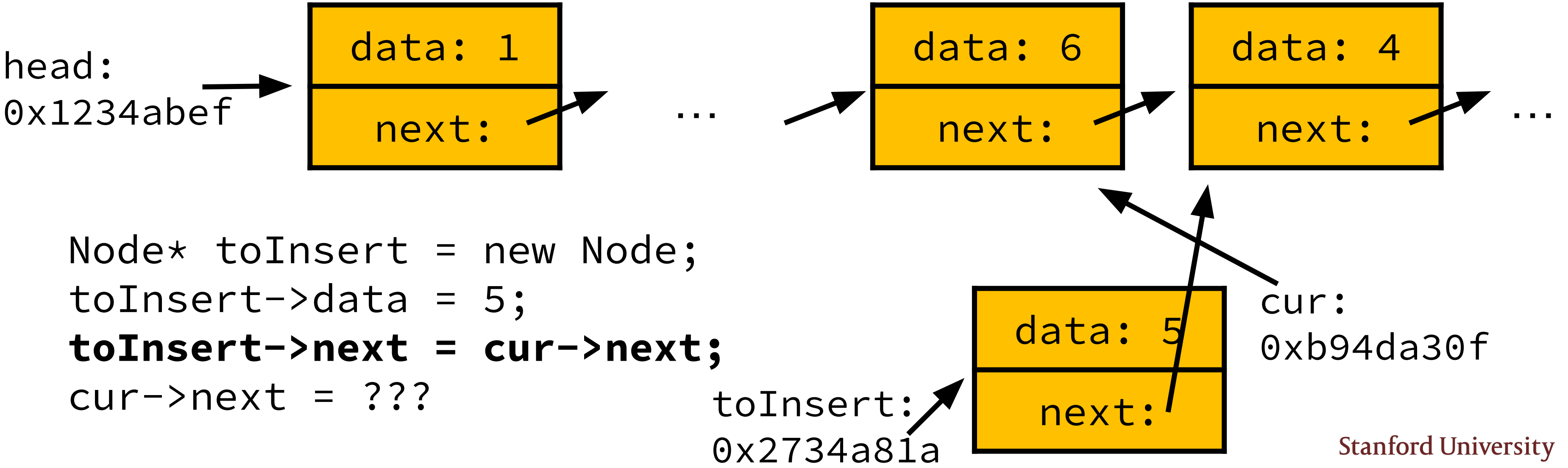



```
Node* toInsert = new Node;
toInsert->data = 5;
toInsert->next = ???
cur->next = ???
```



Linked List Insert

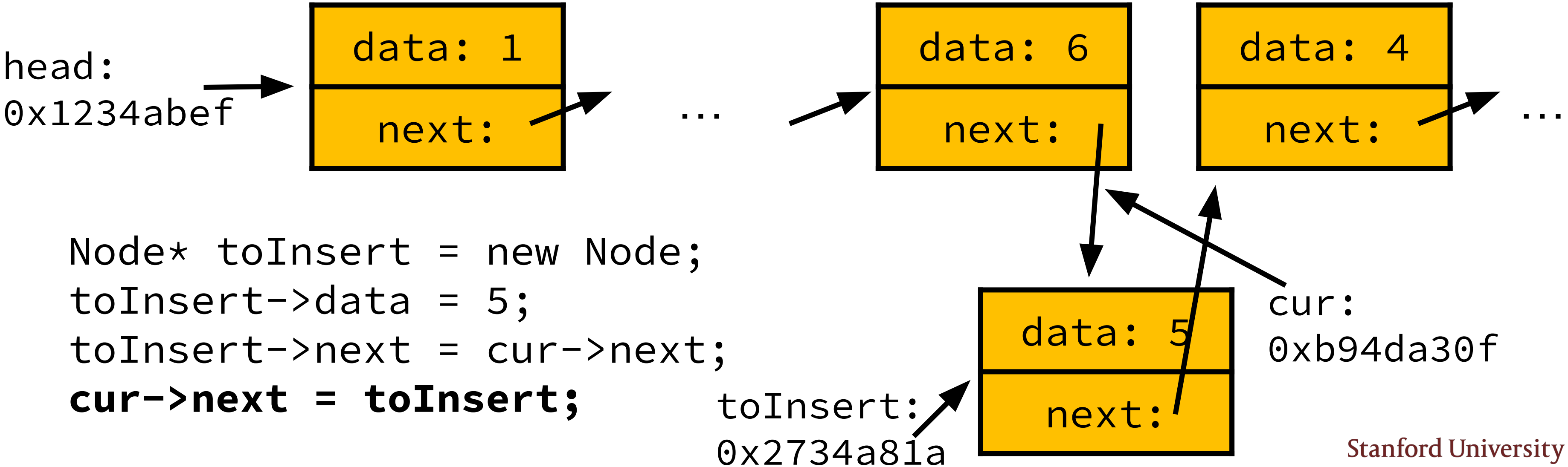
- Traverse to some location, **create and link in new node**
- O(n) - we have to visit O(n) other nodes before reaching location




```
Node* toInsert = new Node;
toInsert->data = 5;
toInsert->next = cur->next;
cur->next = ???
```

Linked List Insert

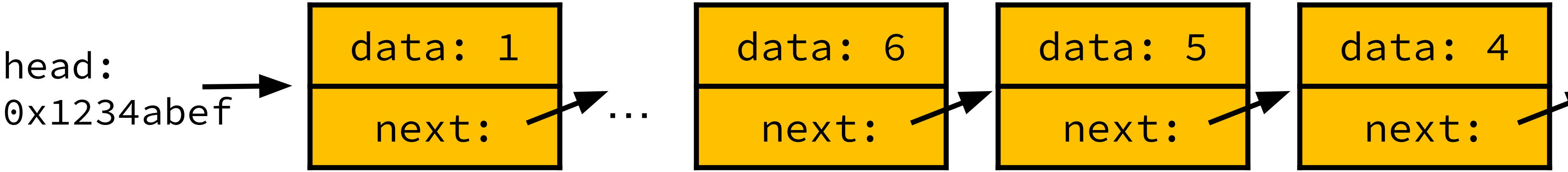
- Traverse to some location, **create and link in new node**
- `O(n)` - we have to visit `O(n)` other nodes before reaching location




```
Node* toInsert = new Node;
toInsert->data = 5;
toInsert->next = cur->next;
cur->next = toInsert;
```

Linked List Insert

- Traverse to some location, **create and link in new node**
- `O(n)` - we have to visit `O(n)` other nodes before reaching location

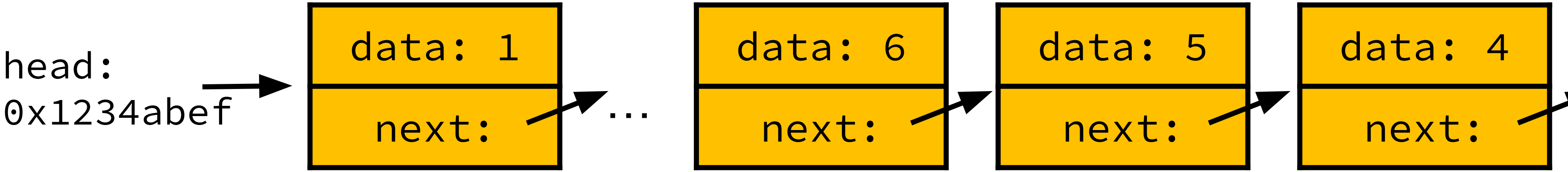



```
Node* toInsert = new Node;
toInsert->data = 5;
toInsert->next = cur->next;
cur->next = toInsert;
```

Linked List Delete

- Traverse to node we want to delete, free AND rewire
- Again, `O(n)`, since it involves linked list traversal

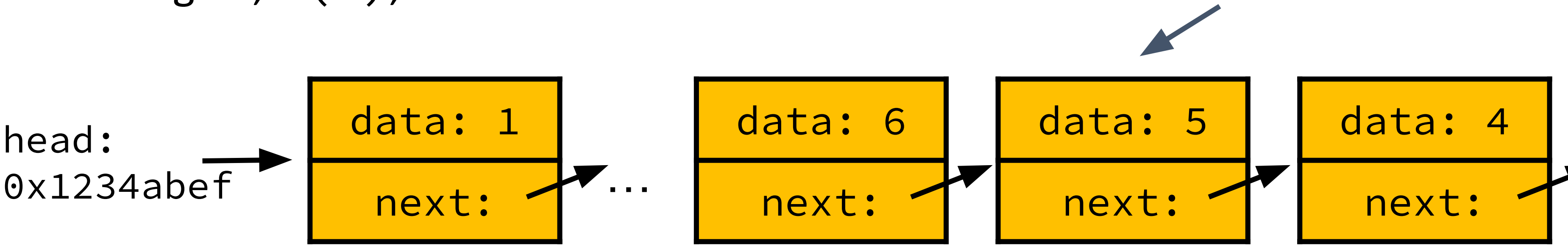
Let's delete this 5 node.



Linked List Delete

- **Traverse to node we want to delete**, free AND rewire
- Again, `O(n)`, since it involves linked list traversal

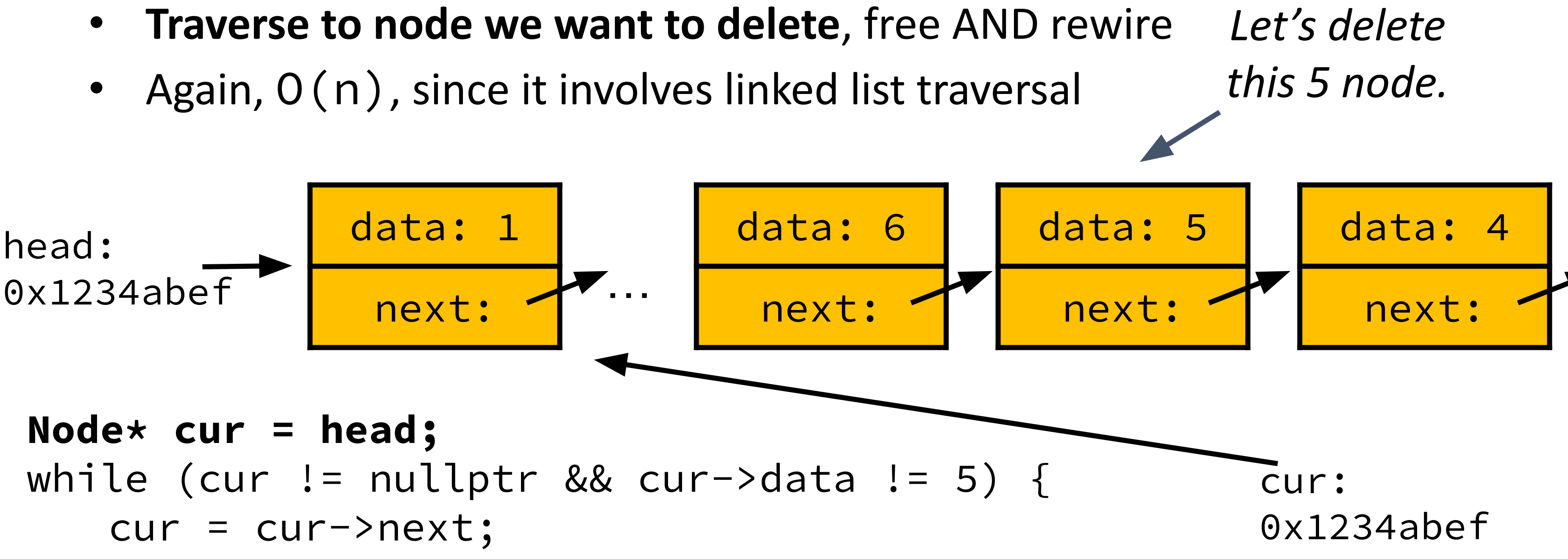
Let's delete this 5 node.




```
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    cur = cur->next;
}
```

Linked List Delete

- **Traverse to node we want to delete**, free AND rewire
- Again, O(n), since it involves linked list traversal

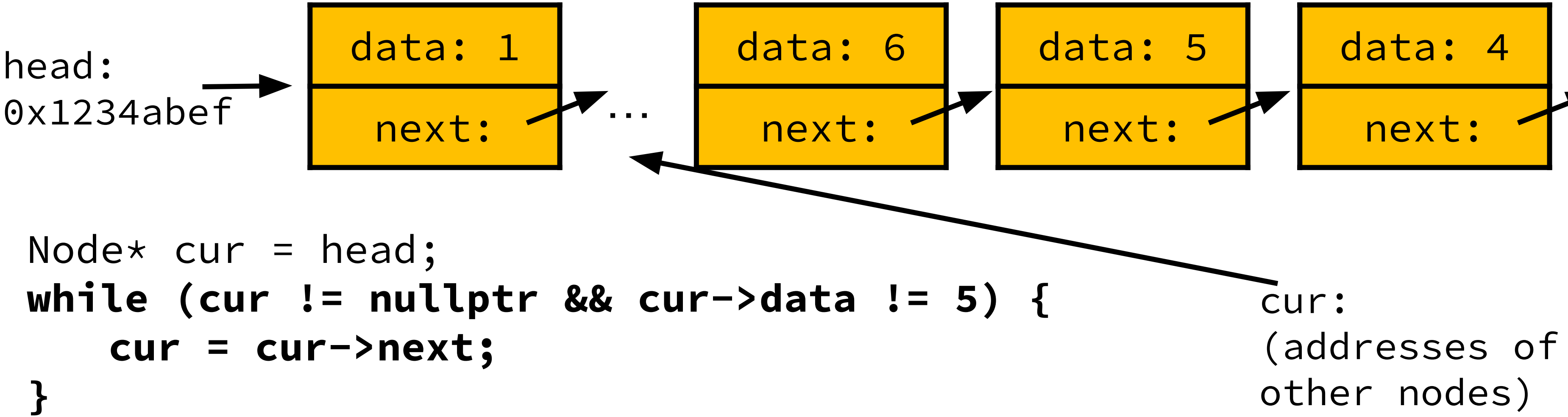



```
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    cur = cur->next;
}
```

Linked List Delete

- **Traverse to node we want to delete**, free AND rewire
- Again, O(n), since it involves linked list traversal

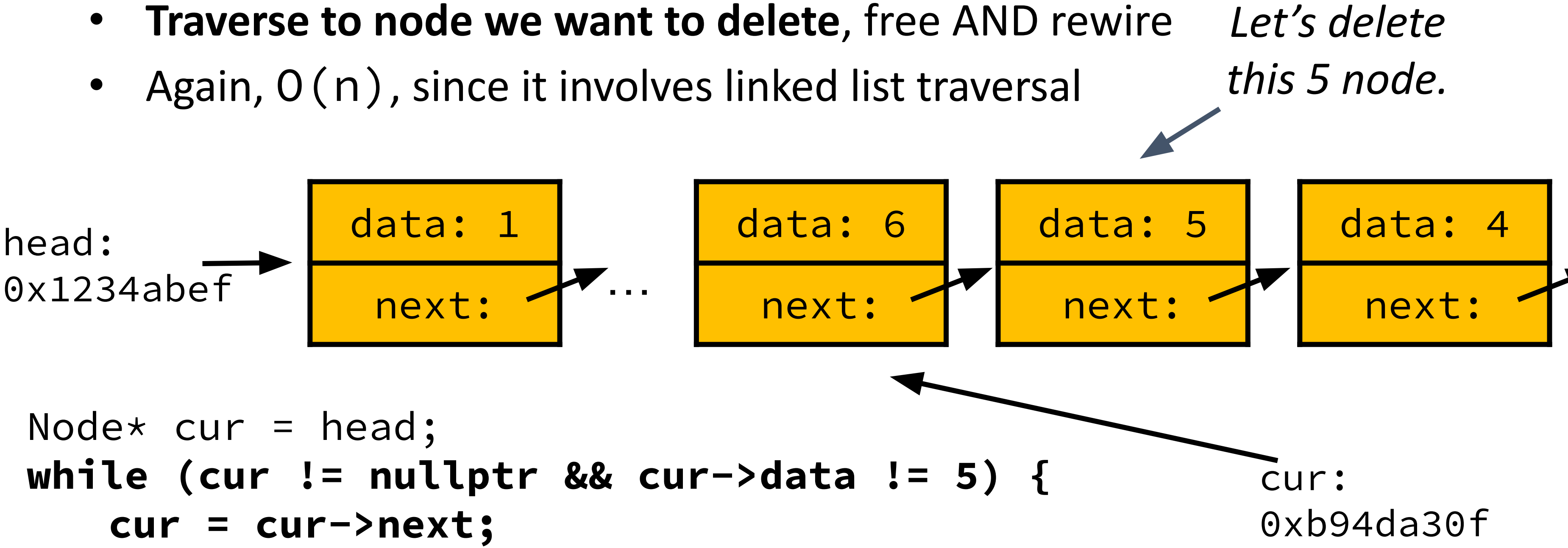
Let's delete this 5 node.




```
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    cur = cur->next;
}
```

Linked List Delete

- **Traverse to node we want to delete**, free AND rewire
- Again, O(n), since it involves linked list traversal

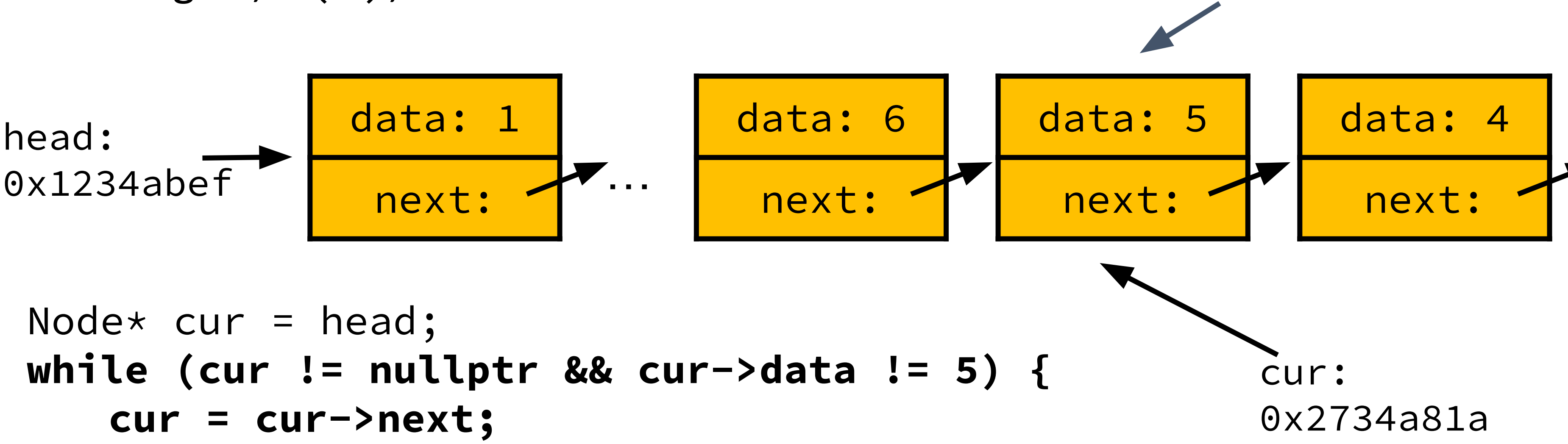



```
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    cur = cur->next;
}
```

Linked List Delete

- **Traverse to node we want to delete**, free AND rewire
- Again, `O(n)`, since it involves linked list traversal

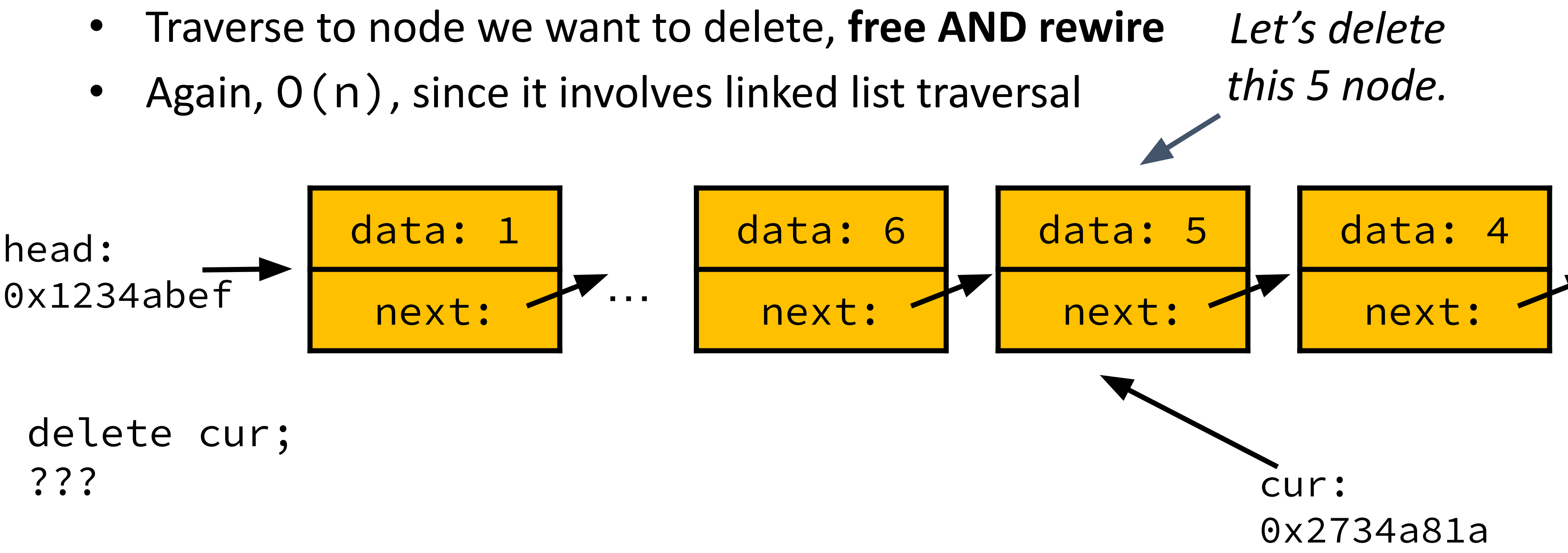
Let's delete this 5 node.




```
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    cur = cur->next;
}
```

Linked List Delete

- Traverse to node we want to delete, **free AND rewire**
- Again, `O(n)`, since it involves linked list traversal

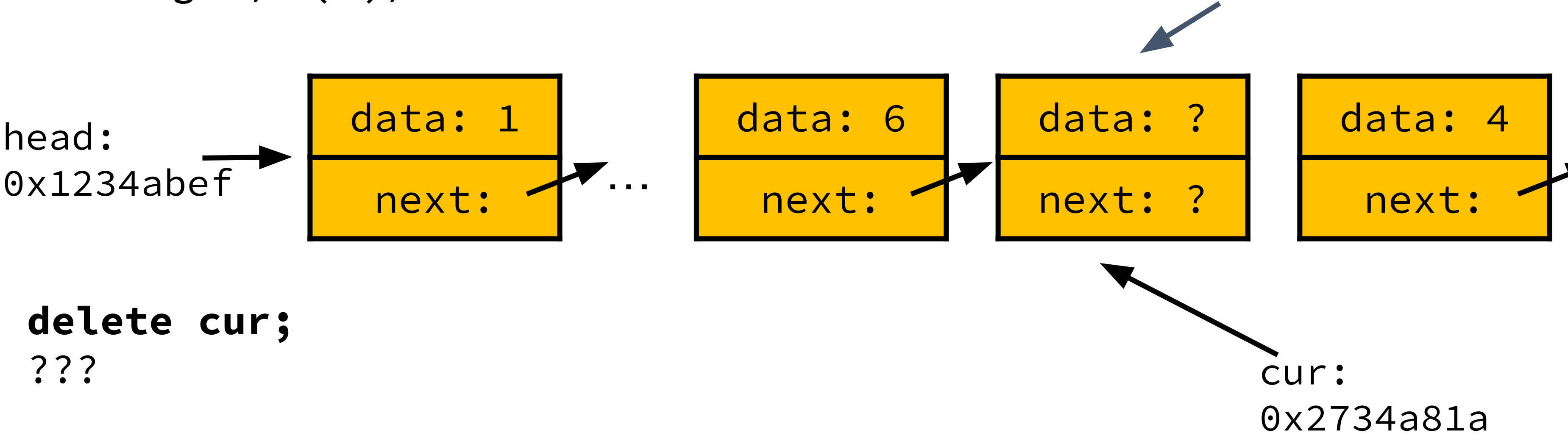



```
delete cur;
???
```

Linked List Delete

- Traverse to node we want to delete, **free AND rewire**
- Again, O(n), since it involves linked list traversal

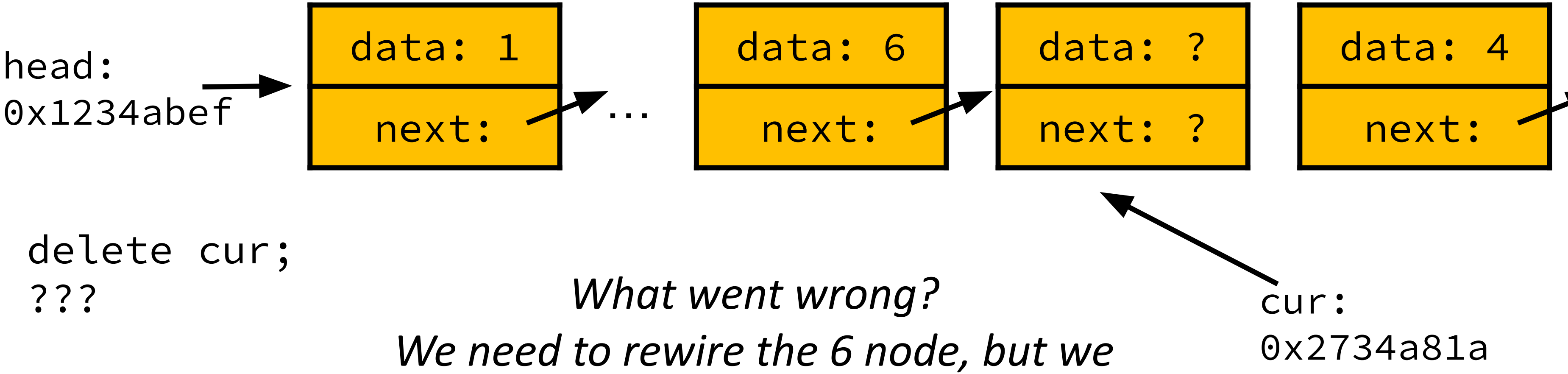
Let's delete this 5 node.



delete cur;
???

Linked List Delete

- Traverse to node we want to delete, **free AND rewire**
- Again, O(n), since it involves linked list traversal




```
delete cur;
???
```

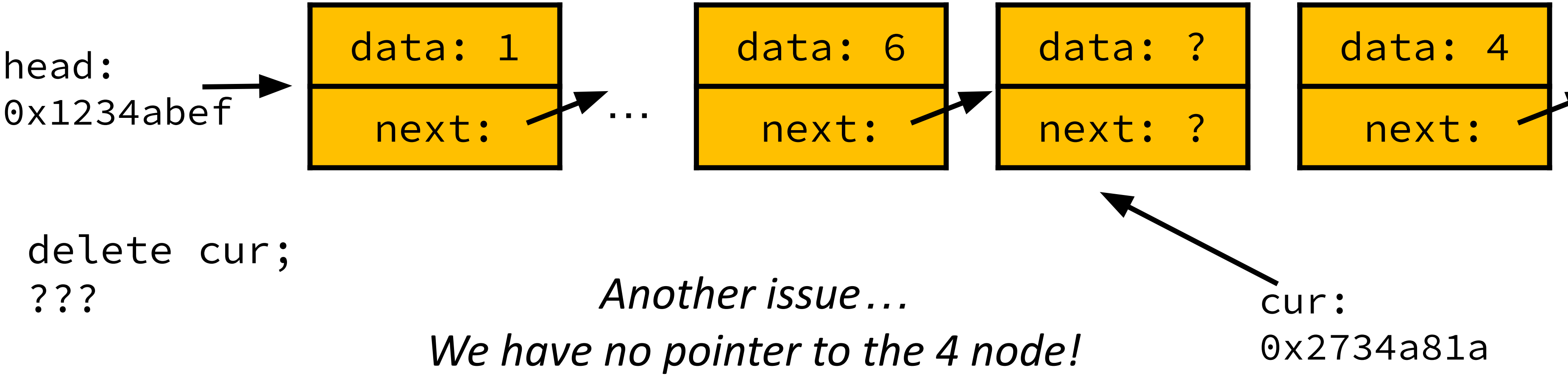
What went wrong?

We need to rewire the 6 node, but we don't have a pointer to it.

Linked List Delete

- Traverse to node we want to delete, **free AND rewire**
- Again, O(n), since it involves linked lis

MEMORY LEAK 👎



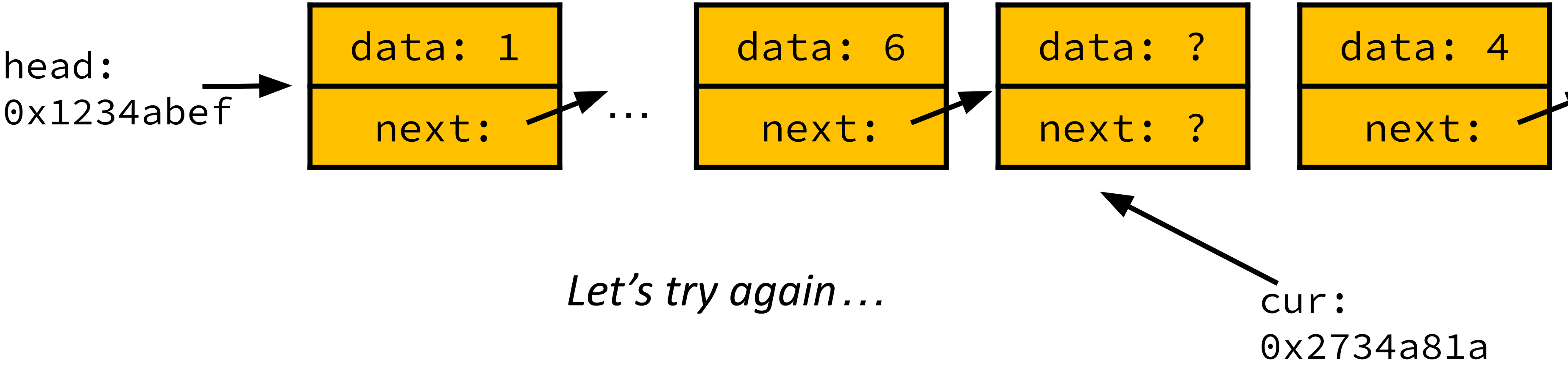

```
delete cur;
???
```

Another issue…

We have no pointer to the 4 node!

Linked List Delete

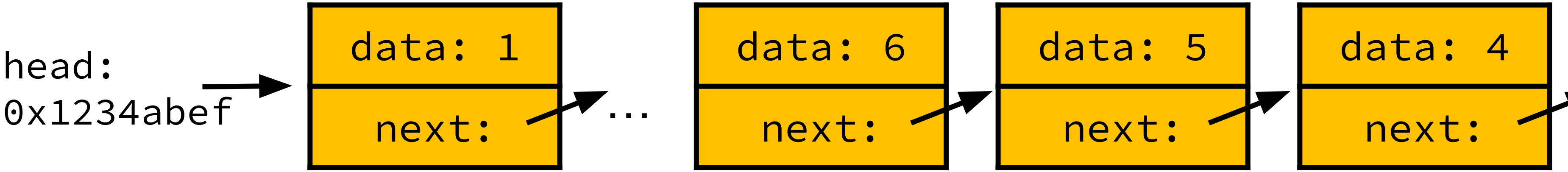
- Traverse to node **before** the one we want to delete, free and rewire
- Again, `O(n)`, since it involves linked list traversal



Let's try again…

Linked List Delete

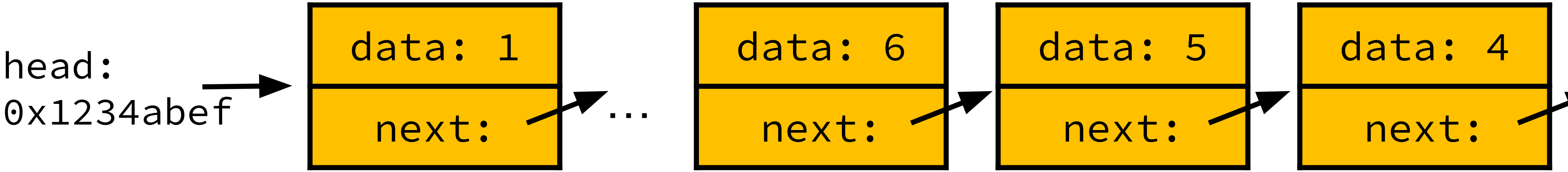
- Traverse to node **before** the one we want to delete, free and rewire
- Again, O(n), since it involves linked list traversal




```
Node* prev = nullptr;
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    prev = cur;
    cur = cur->next;
}
```

Linked List Delete

- Traverse to node **before** the one we want to delete, free and rewire
- Again, O(n), since it involves linked list traversal

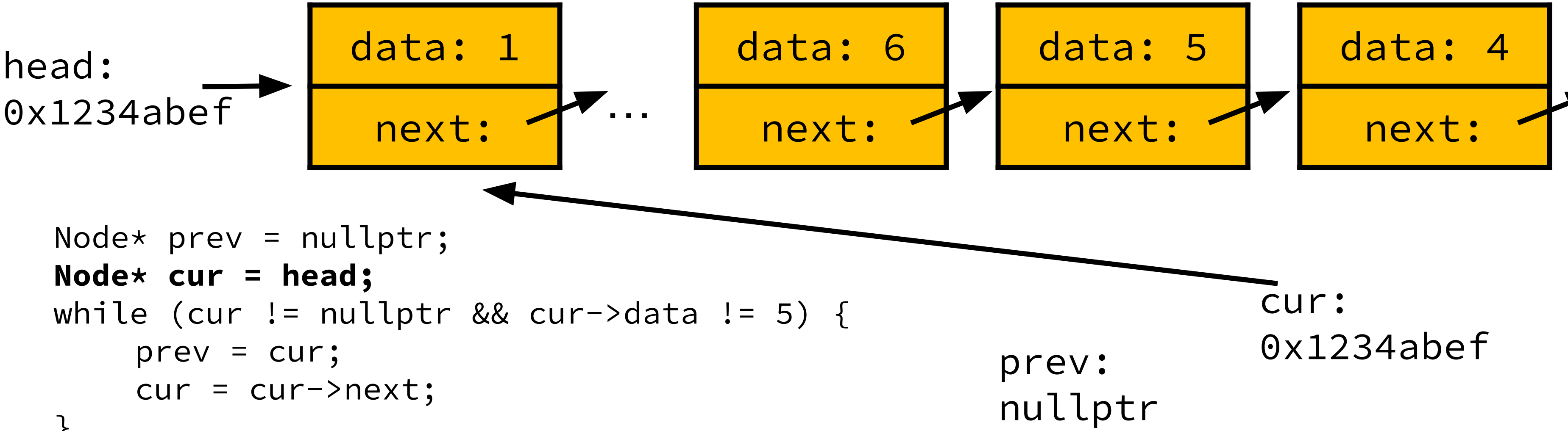



```
Node* prev = nullptr;
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    prev = cur;
    cur = cur->next;
}
```

prev:
nullptr

Linked List Delete

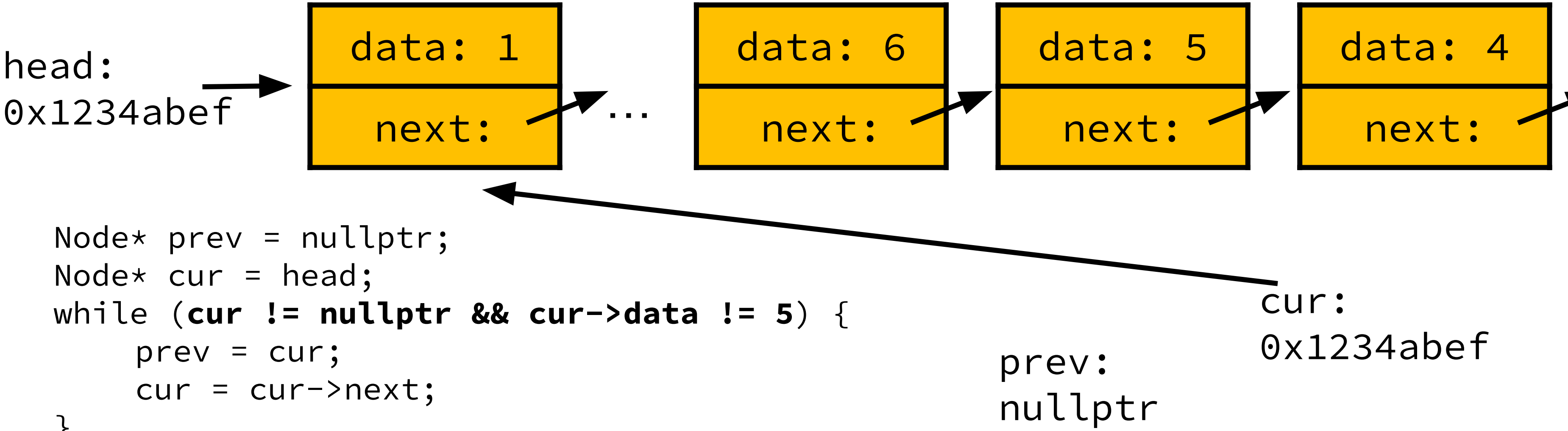
- Traverse to node **before** the one we want to delete, free and rewire
- Again, O(n), since it involves linked list traversal




```
Node* prev = nullptr;
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    prev = cur;
    cur = cur->next;
}
```

Linked List Delete

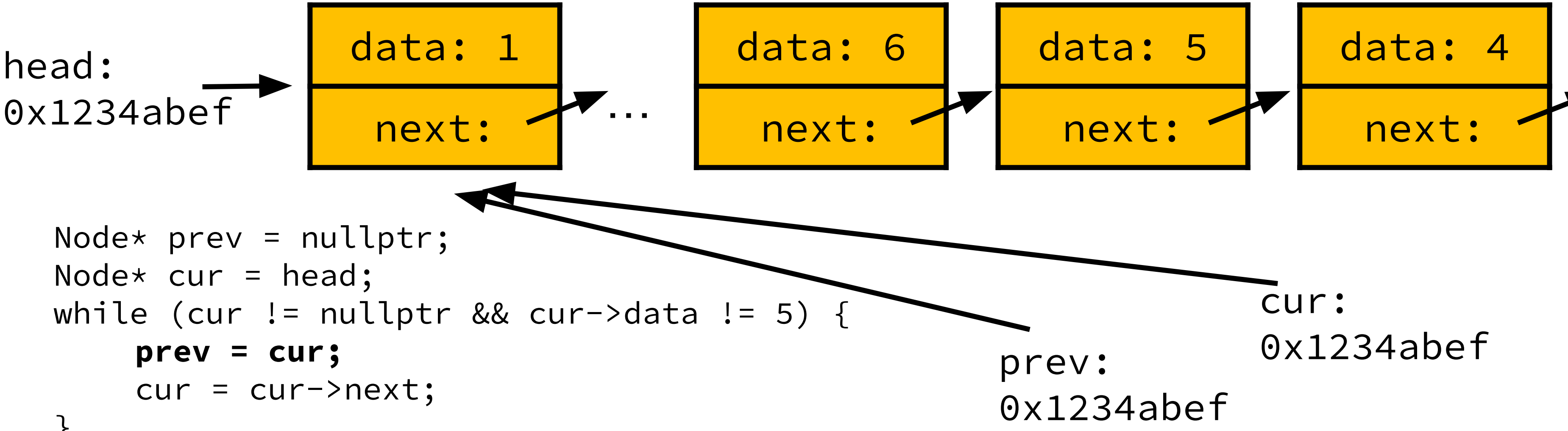
- Traverse to node **before** the one we want to delete, free and rewire
- Again, O(n), since it involves linked list traversal




```
Node* prev = nullptr;
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    prev = cur;
    cur = cur->next;
}
```

Linked List Delete

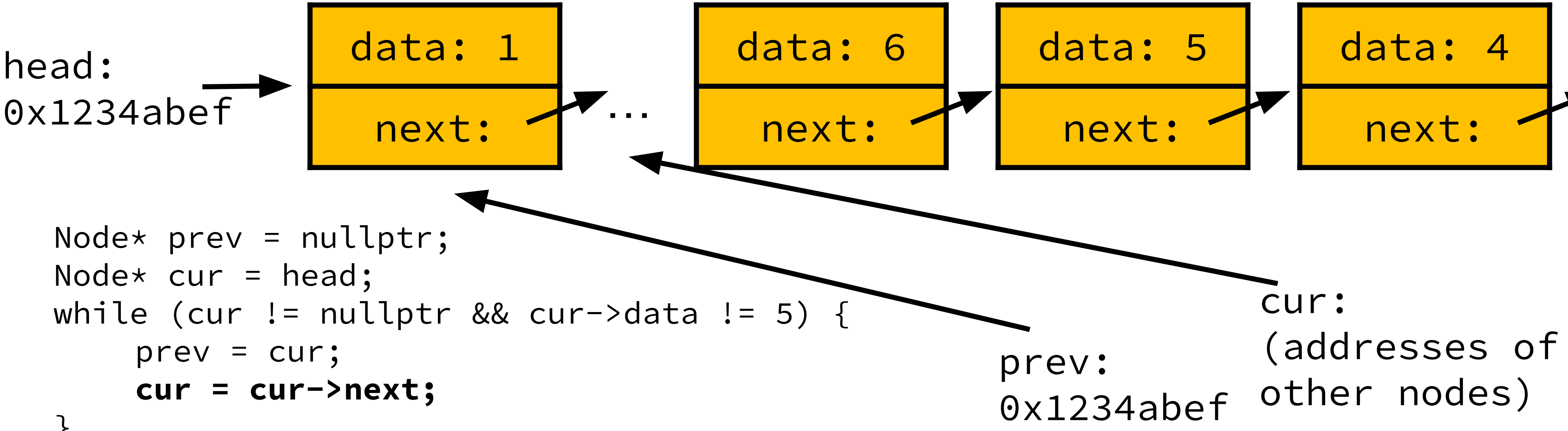
- Traverse to node **before** the one we want to delete, free and rewire
- Again, O(n), since it involves linked list traversal




```
Node* prev = nullptr;
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    prev = cur;
    cur = cur->next;
}
```

Linked List Delete

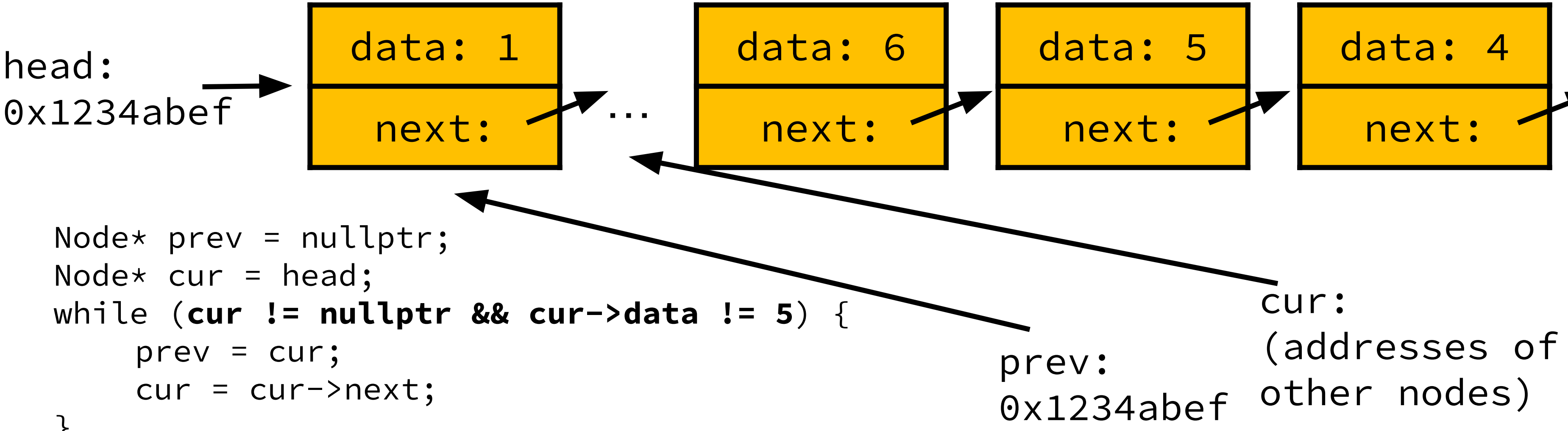
- Traverse to node **before** the one we want to delete, free and rewire
- Again, O(n), since it involves linked list traversal




```
Node* prev = nullptr;
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    prev = cur;
    cur = cur->next;
}
```

Linked List Delete

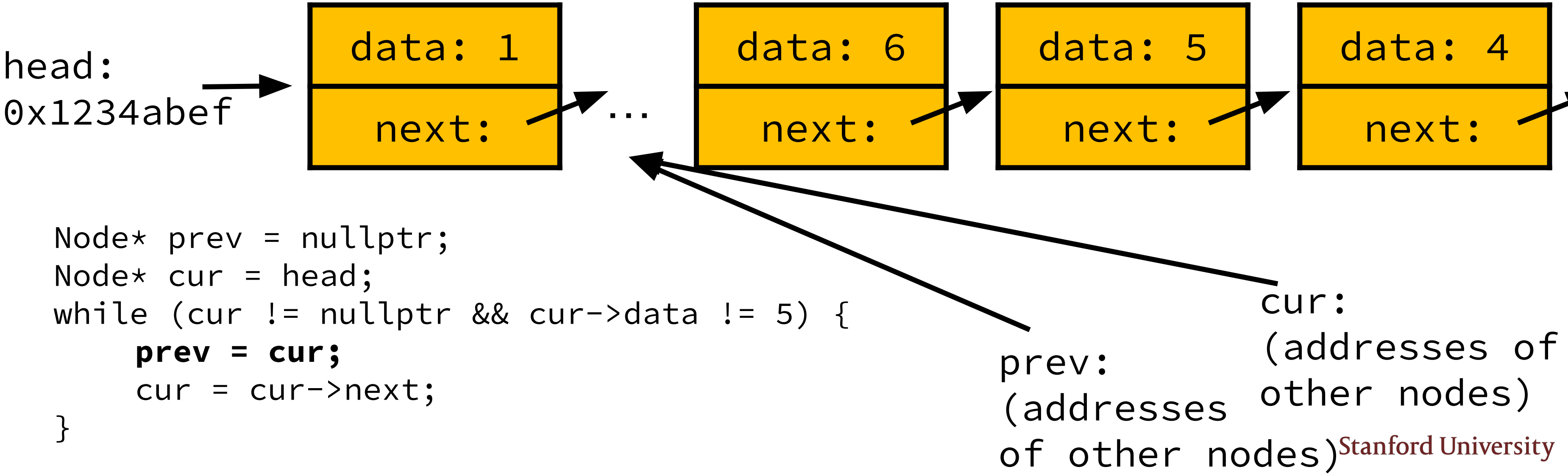
- Traverse to node **before** the one we want to delete, free and rewire
- Again, O(n), since it involves linked list traversal




```
Node* prev = nullptr;
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    prev = cur;
    cur = cur->next;
}
```

Linked List Delete

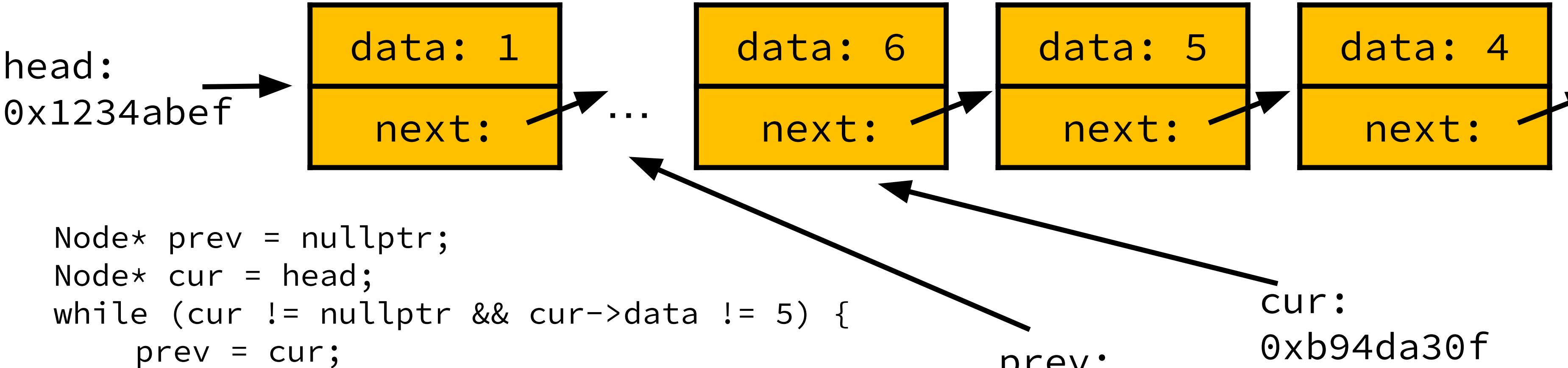
- Traverse to node **before** the one we want to delete, free and rewire
- Again, O(n), since it involves linked list traversal




```
Node* prev = nullptr;
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    prev = cur;
    cur = cur->next;
}
```

Linked List Delete

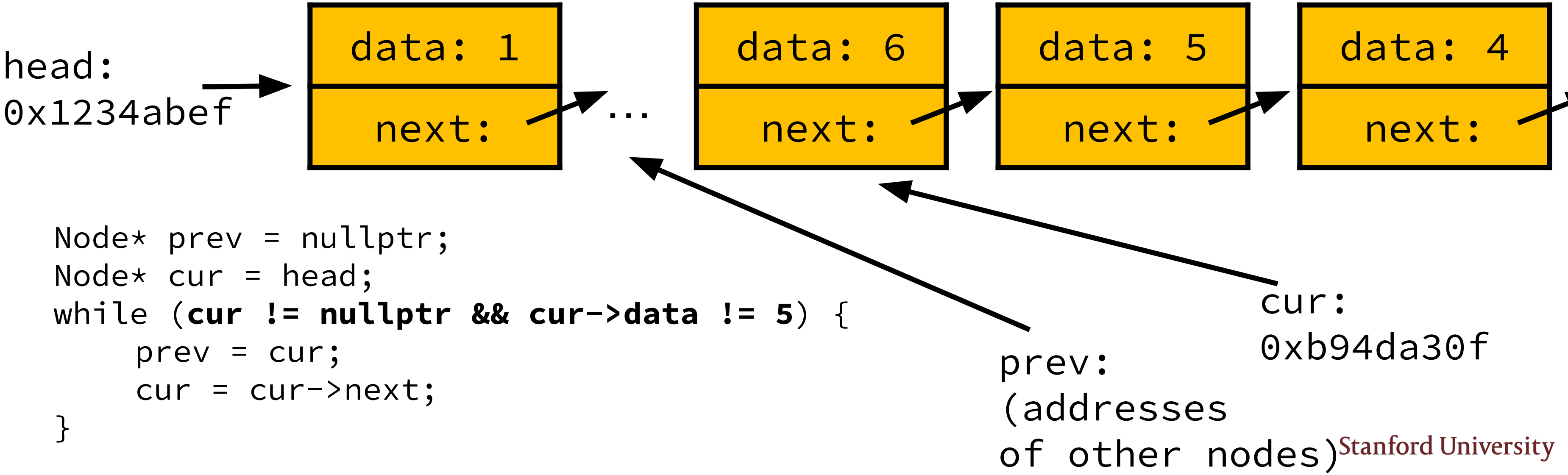
- Traverse to node **before** the one we want to delete, free and rewire
- Again, `O(n)`, since it involves linked list traversal




```
Node* prev = nullptr;
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    prev = cur;
    cur = cur->next;
}
```

Linked List Delete

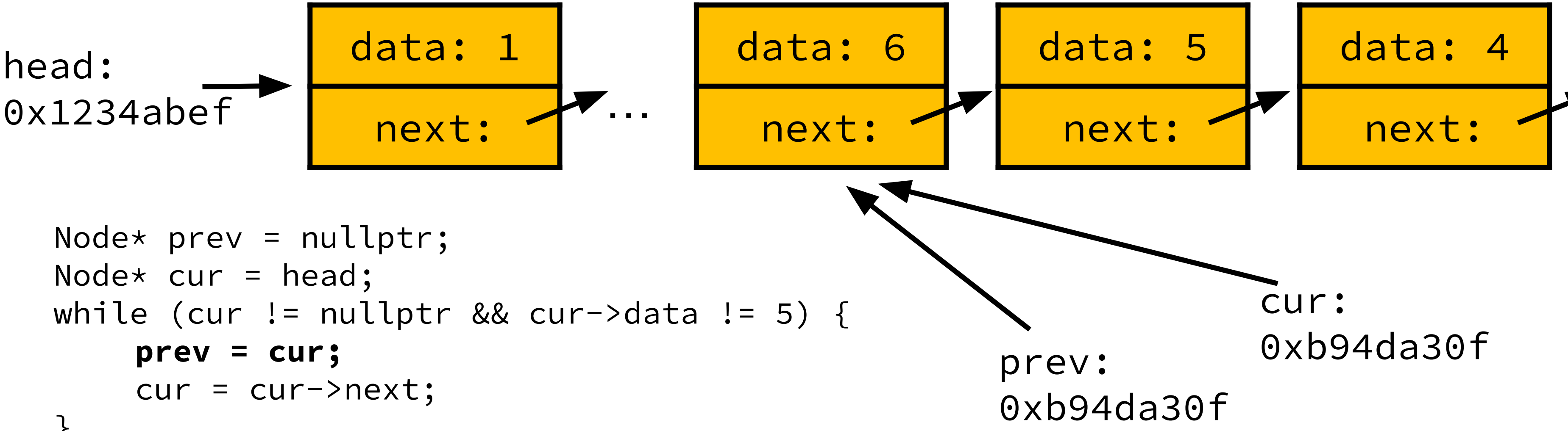
- Traverse to node **before** the one we want to delete, free and rewire
- Again, O(n), since it involves linked list traversal




```
Node* prev = nullptr;
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    prev = cur;
    cur = cur->next;
}
```

Linked List Delete

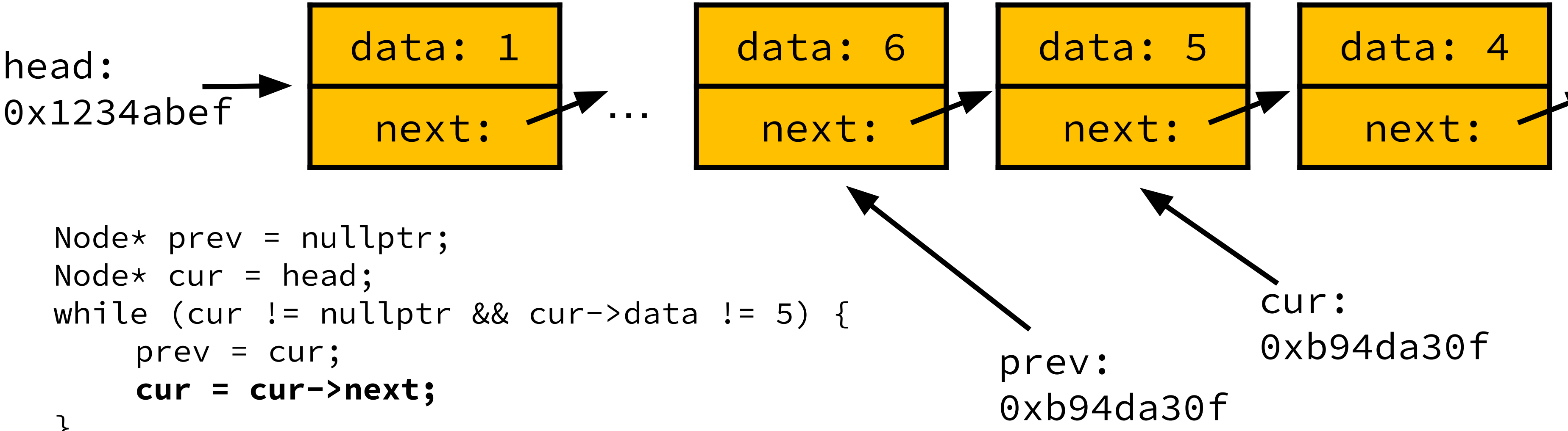
- Traverse to node **before** the one we want to delete, free and rewire
- Again, O(n), since it involves linked list traversal




```
Node* prev = nullptr;
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    prev = cur;
    cur = cur->next;
}
```

Linked List Delete

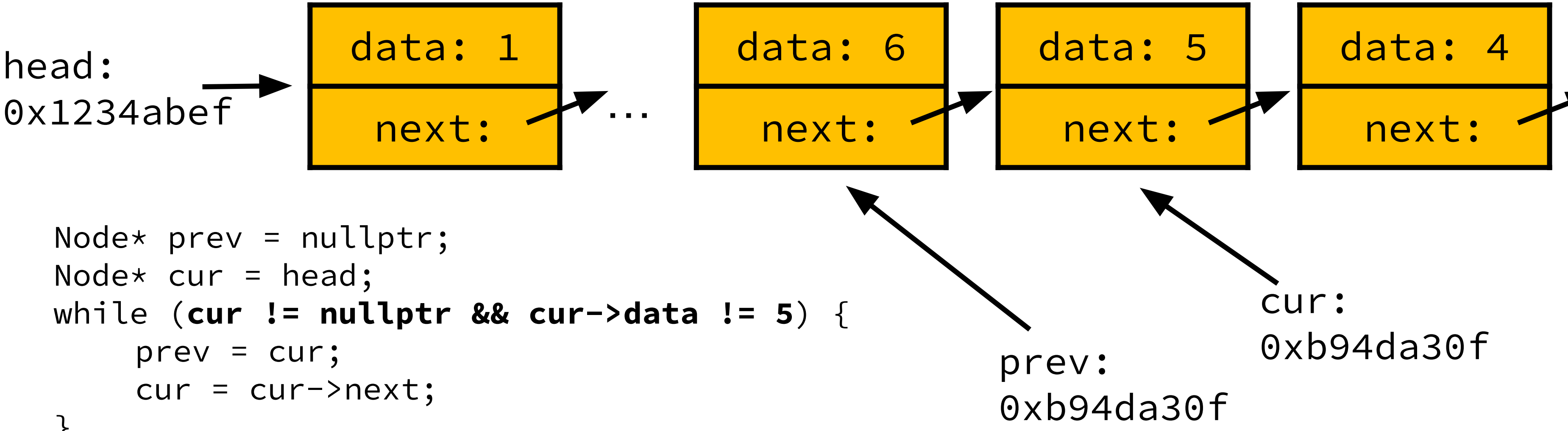
- Traverse to node **before** the one we want to delete, free and rewire
- Again, O(n), since it involves linked list traversal




```
Node* prev = nullptr;
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    prev = cur;
    cur = cur->next;
}
```

Linked List Delete

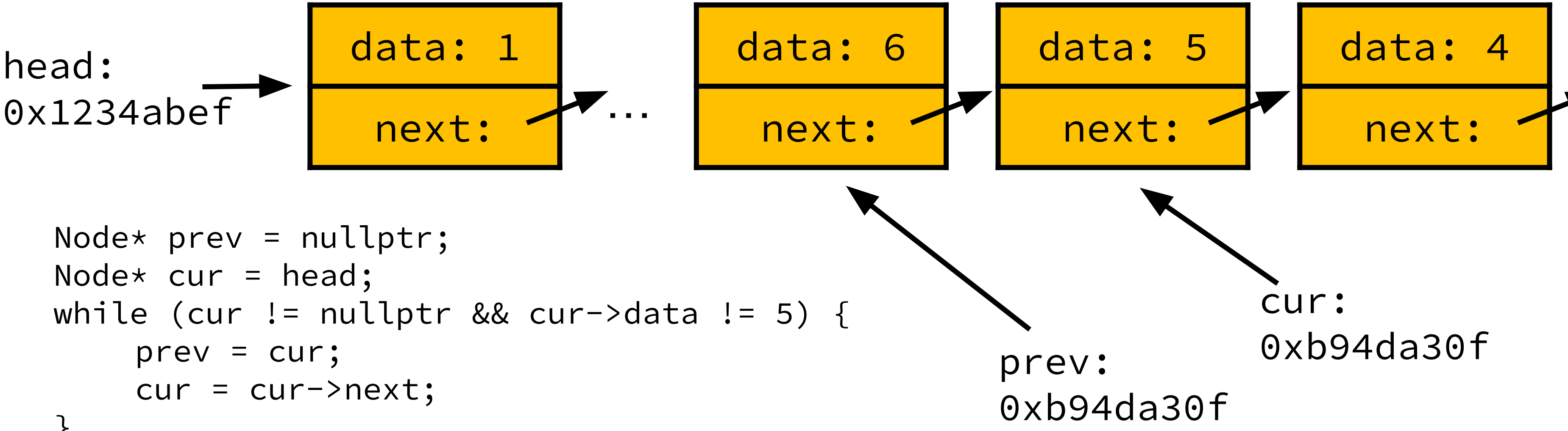
- Traverse to node **before** the one we want to delete, free and rewire
- Again, O(n), since it involves linked list traversal




```
Node* prev = nullptr;
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    prev = cur;
    cur = cur->next;
}
```

Linked List Delete

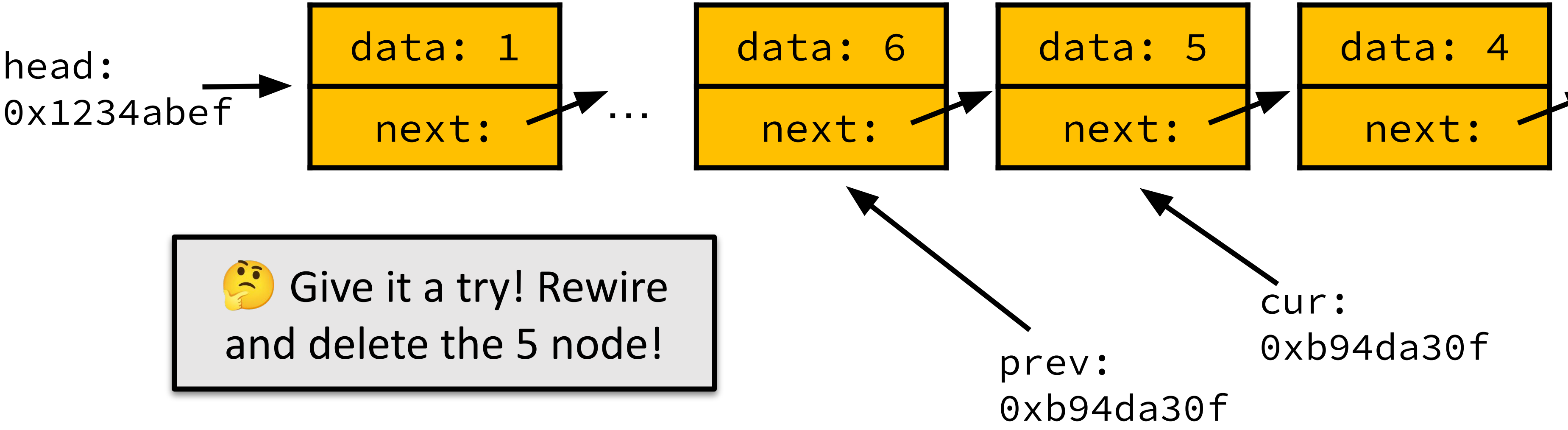
- Traverse to node **before** the one we want to delete, free and rewire
- Again, O(n), since it involves linked list traversal




```
Node* prev = nullptr;
Node* cur = head;
while (cur != nullptr && cur->data != 5) {
    prev = cur;
    cur = cur->next;
}
```

Linked List Delete

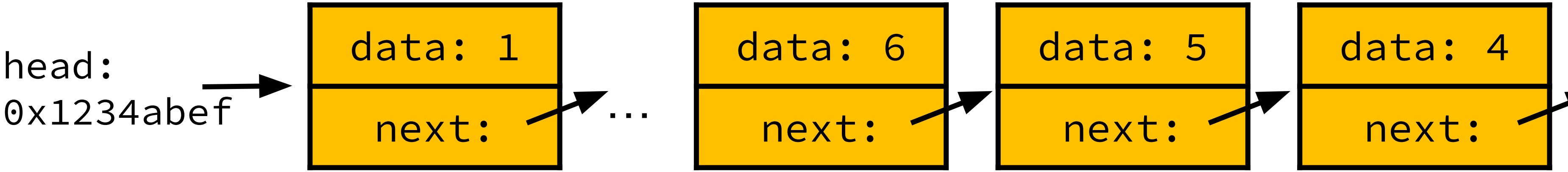
- Traverse to node before the one we want to delete, **free and rewire**
- Again, `O(n)`, since it involves linked list traversal



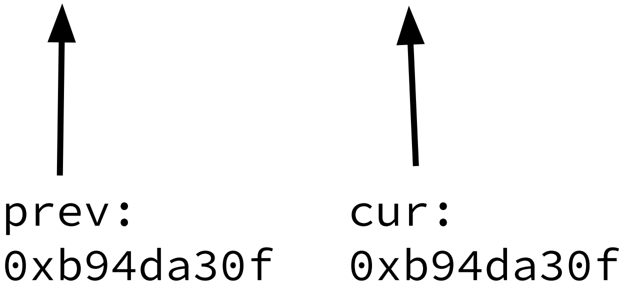
🤔 Give it a try! Rewire and delete the 5 node!

Linked List Delete

- Traverse to node before the one we want to delete, **free and rewire**
- Again, `O(n)`, since it involves linked list traversal

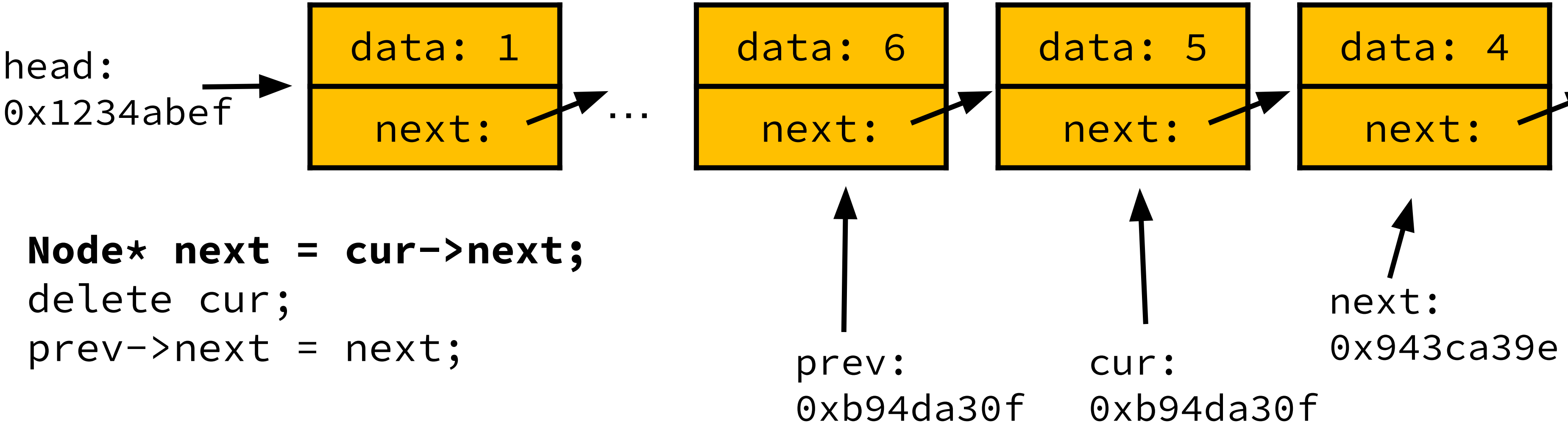



```
Node* next = cur->next;
delete cur;
prev->next = next;
```



Linked List Delete

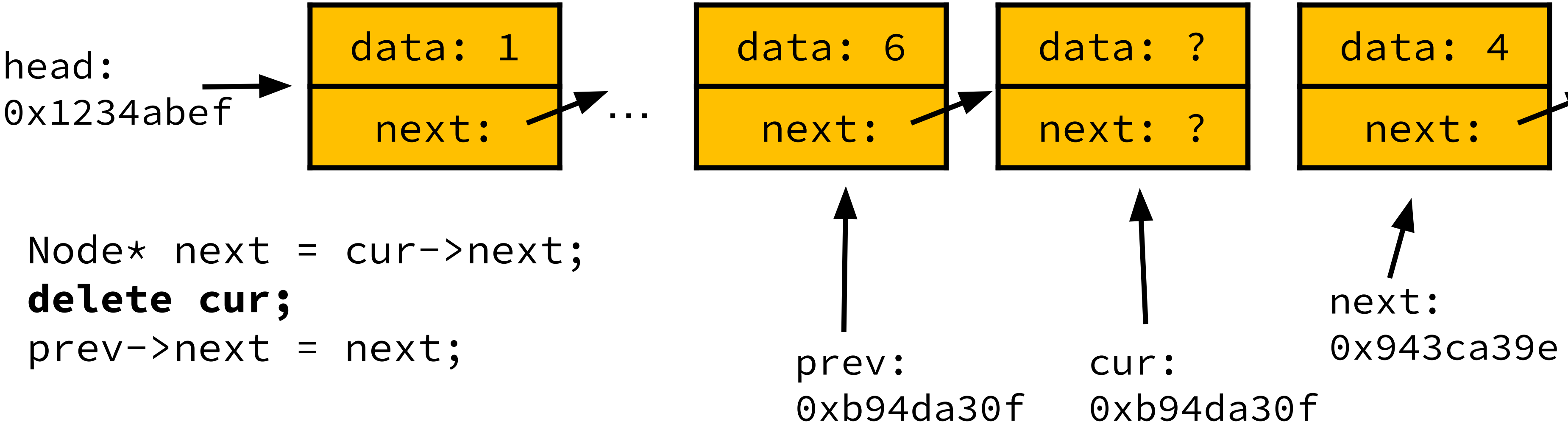
- Traverse to node before the one we want to delete, **free and rewire**
- Again, O(n), since it involves linked list traversal




```
Node* next = cur->next;
delete cur;
prev->next = next;
```

Linked List Delete

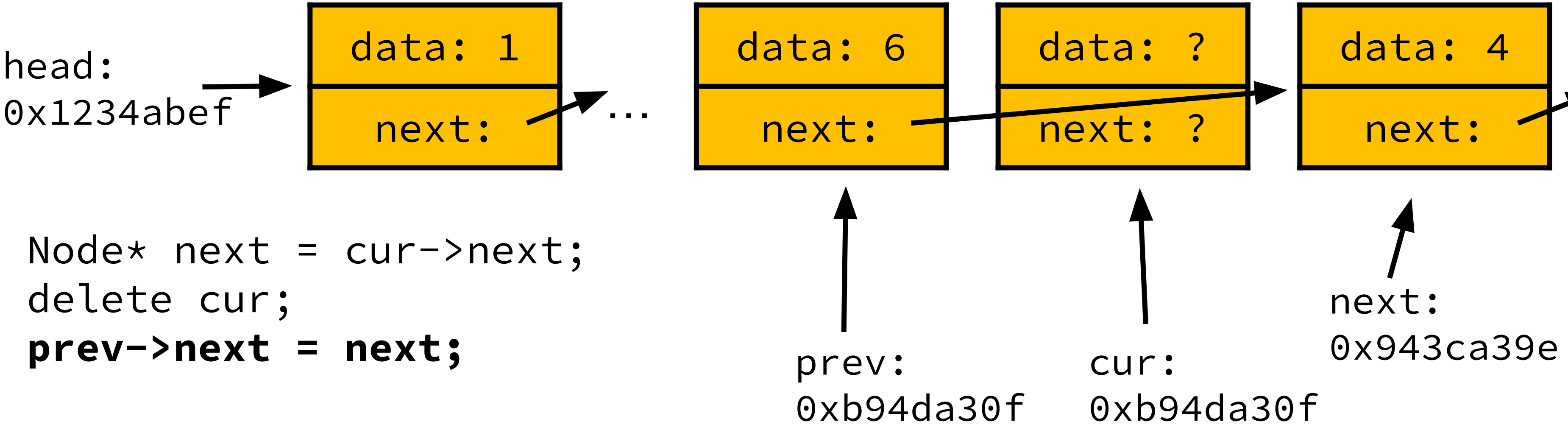
- Traverse to node before the one we want to delete, **free and rewire**
- Again, `O(n)`, since it involves linked list traversal




```
Node* next = cur->next;
delete cur;
prev->next = next;
```

Linked List Delete

- Traverse to node before the one we want to delete, **free and rewire**
- Again, O(n), since it involves linked list traversal

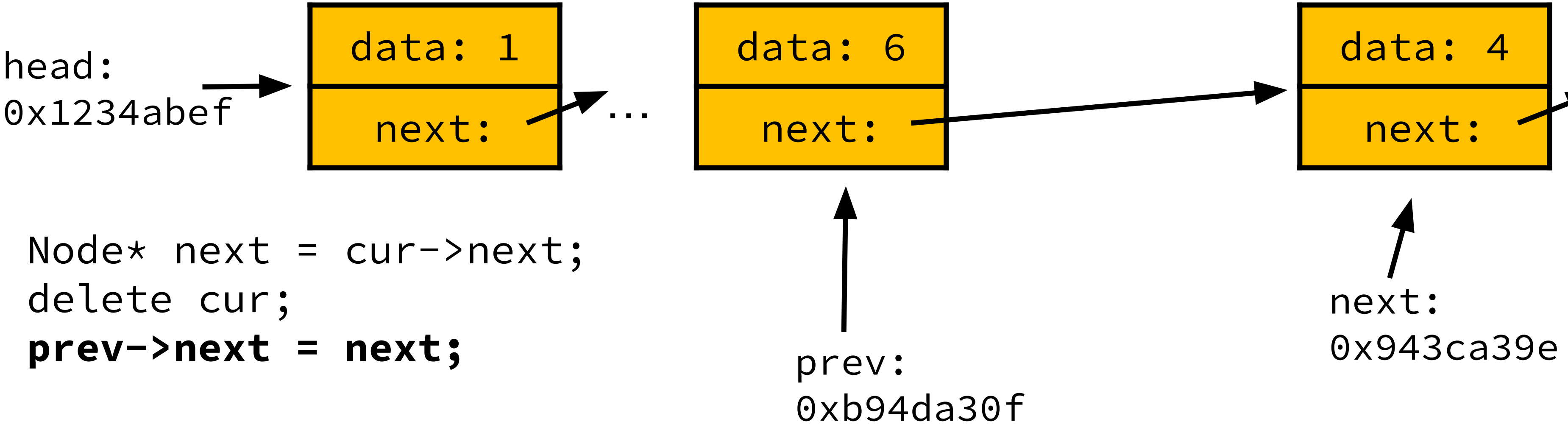



```
Node* next = cur->next;
delete cur;
prev->next = next;
```

Linked List Delete

HAPPY TIMES 👍

- Traverse to node before the one we want to delete, **free and rewire**
- Again, O(n), since it involves linked list traversal




```
Node* next = cur->next;
delete cur;
prev->next = next;
```

Demo: `deleteNode`

Implement delete as described in previous slides

Solution

```
void deleteNode(Node*& list, int value) {
    // traverse to node before value to delete
    Node* prev = nullptr;
    Node* cur = list;
    while (cur != nullptr && cur->data != value) {
        prev = cur;
        cur = cur->next;
    }
    // delete and rewire
    Node* next = cur->next;
    delete cur;
    if (prev != nullptr) {  // added this
        prev->next = next;
    } else {
        list = next;        // and this
    }
}
```

Linked Lists vs. Arrays, Big-O

Linked Lists

- Prepend - `O(1)`
- Append - `O(n)`
- Insert - `O(n)`
- Delete - `O(n)`
- Traverse - `O(n)`

Arrays

- Prepend - `O(n)`
- Append - `O(1)`
- Insert - `O(n)`
- Delete - `O(n)`
- Traverse - `O(n)`

Linked Lists vs. Arrays, Big-O

Linked Lists

- Prepend - `O(1)`
- **Append - `O(n)`**
- Insert - `O(n)`
- Delete - `O(n)`
- Traverse - `O(n)`

Arrays

- Prepend - `O(n)`
- Append - `O(1)`
- Insert - `O(n)`
- Delete - `O(n)`
- Traverse - `O(n)`

This isn't great…

Could we store a pointer to the tail of our list?

Demo: `createList`

Create a linked list from user input

Solution $O(n^2)$

```
Node* createListWithAppend() {
    Node* list = nullptr;
    while (true) {
        int value = getInteger("Next value: ");
        if (value == 0) break;
        appendTo(list, value);
    }
    return list;
}
```

Solution O(n)

```
Node* createListWithTailPtr() {
    Node* head = nullptr;
    Node* tail = head;
    while (true) {
        int value = getInteger("Next value: ");
        if (value == 0) break;
        if (head == nullptr) {
            head = new Node(value, nullptr);
            tail = head;
        } else {
            Node* nextNode = new Node(value, nullptr);
            tail->next = nextNode;
            tail = nextNode;
        }
    }
    return head;
}
```

Passing Pointers by Value

- Unless specified otherwise, parameters in C++ are passed by value – this includes pointers!
- When passed by value, callee function gets a copy of the pointer; **it cannot change where the original pointer points**

Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

```
head: nullptr
```

Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

```
head: nullptr
```

Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

```
head: nullptr
data: 5
```

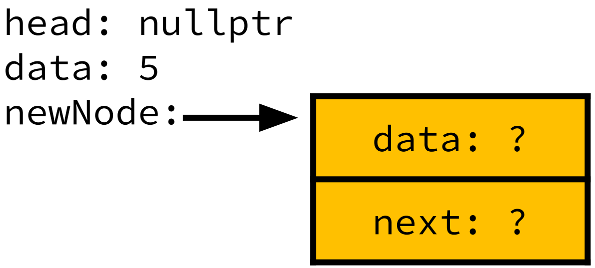
```
head: nullptr
```

Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```



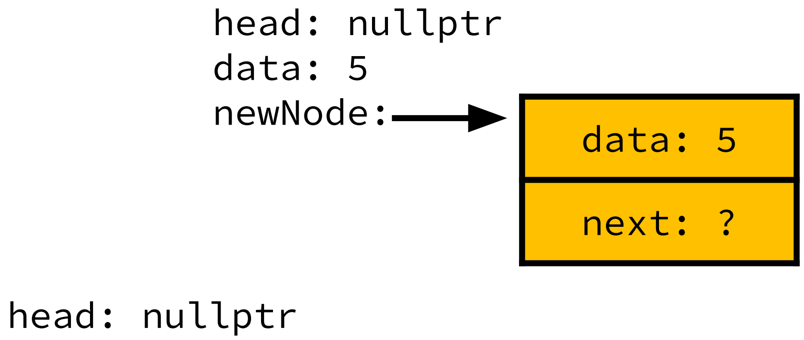
head: nullptr

Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

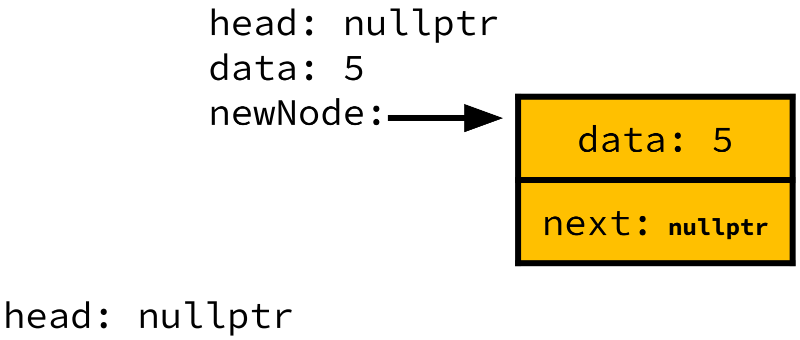


Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

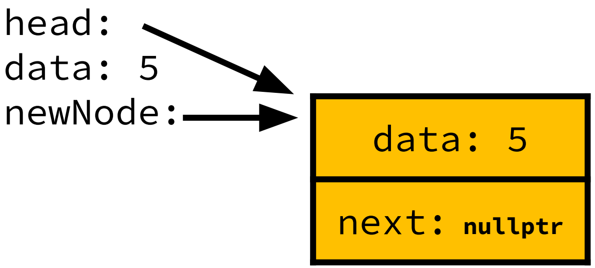


Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```



head: nullptr

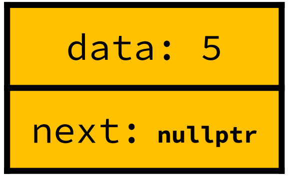
Note: this was a copy of the original head, so head from main doesn't get changed!

Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```



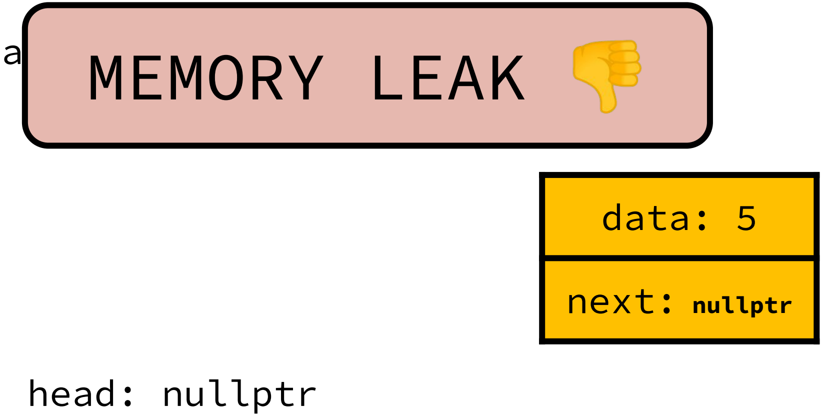
head: nullptr

Passing Pointers by Value

- When passed by value, callee function gets a copy of the pointer; it cannot change where the original pointer points

```
void prependTo(Node* head, int data
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```



Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

```
head: nullptr
```

Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

```
head: nullptr
```

Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

```
head: nullptr
data: 5
```

```
head: nullptr
```

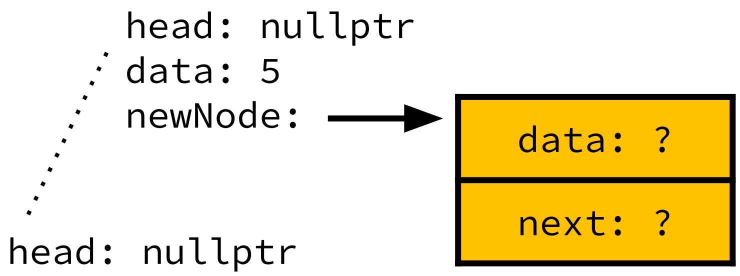
Note: we didn't make a copy of head, `prependTo` gets access to the head variable from back in `main`!

Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

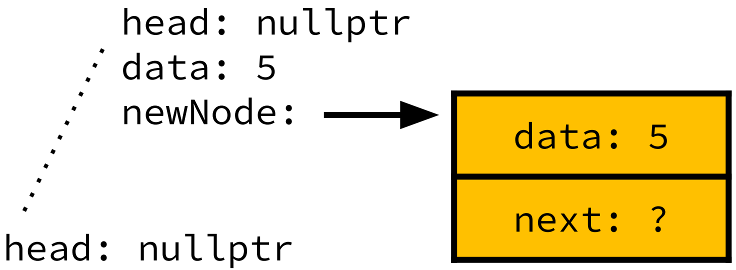


Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

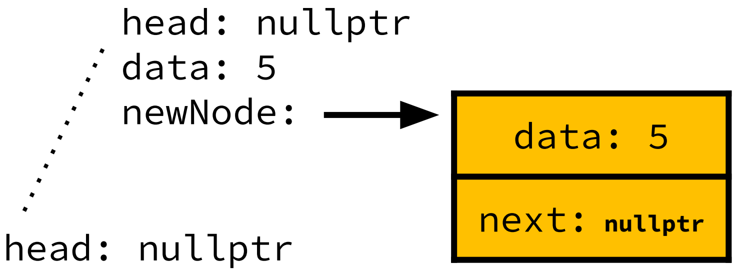


Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

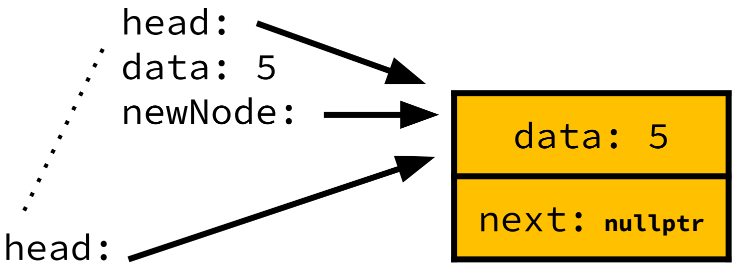


Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

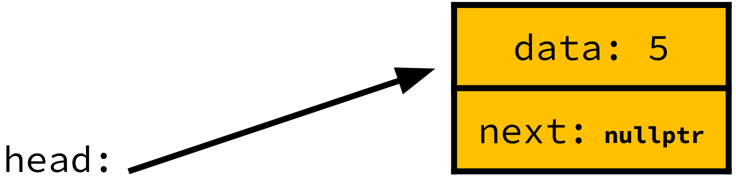


Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

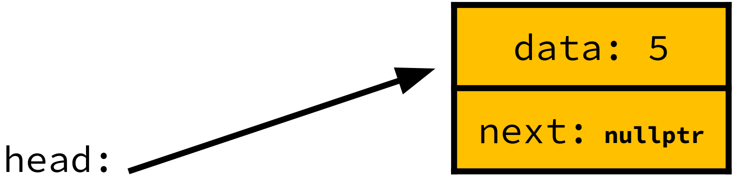


Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```



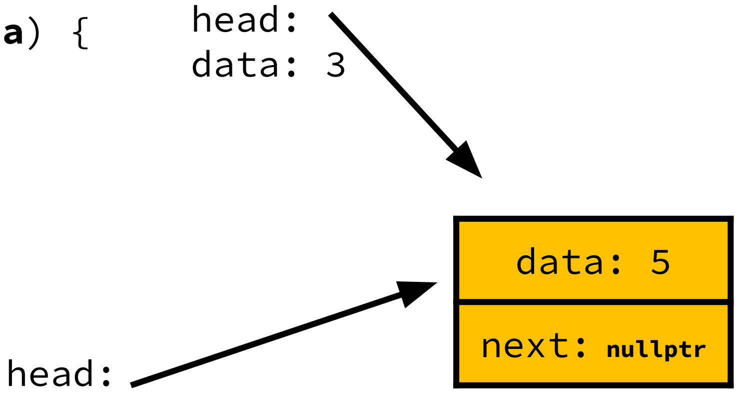
👥 *Trace the next function call with a neighbor!*

Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

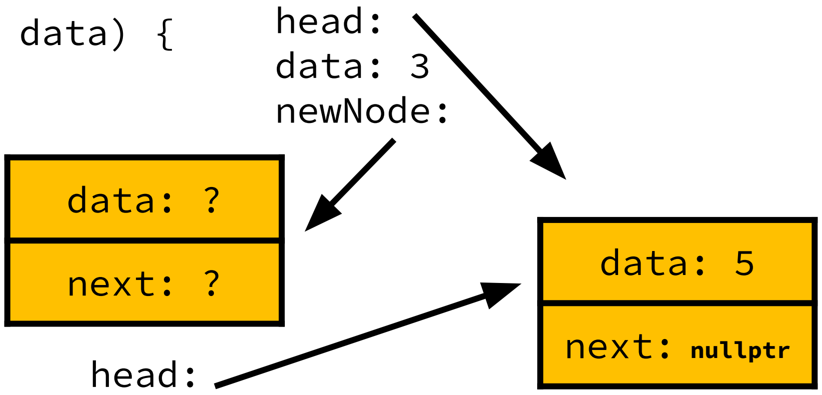


Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

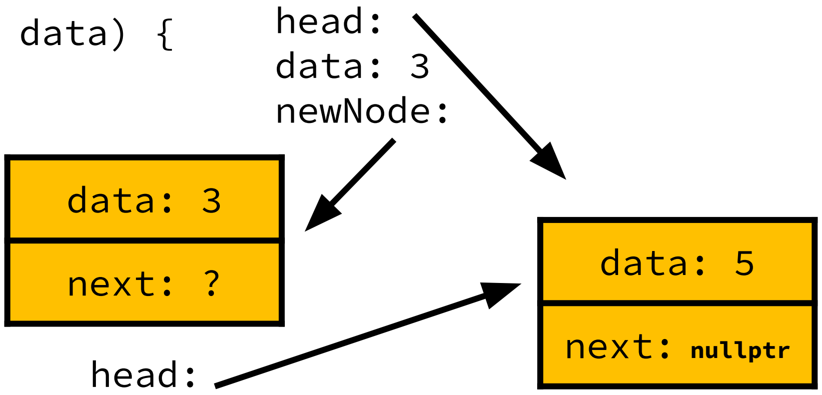


Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

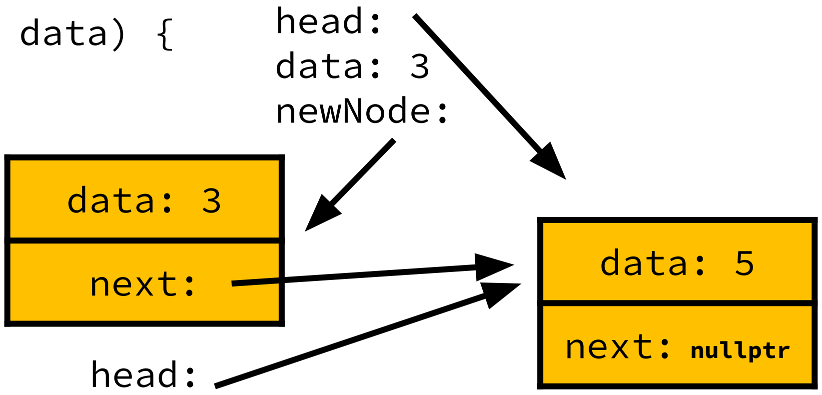


Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

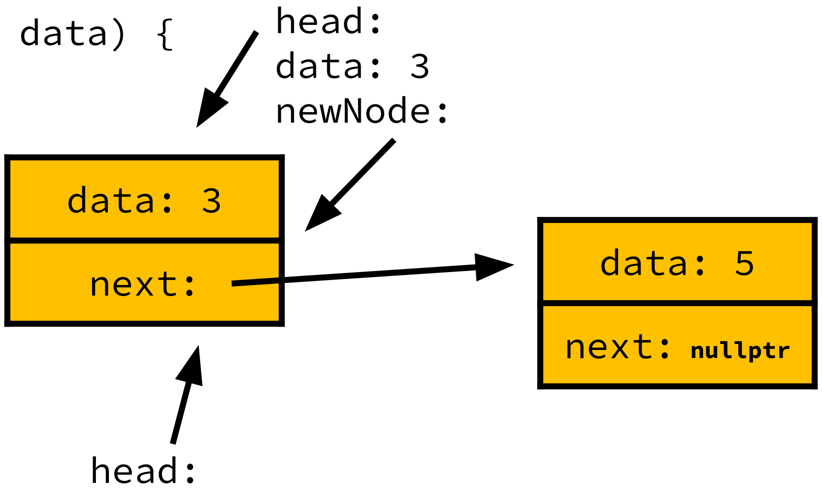


Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```

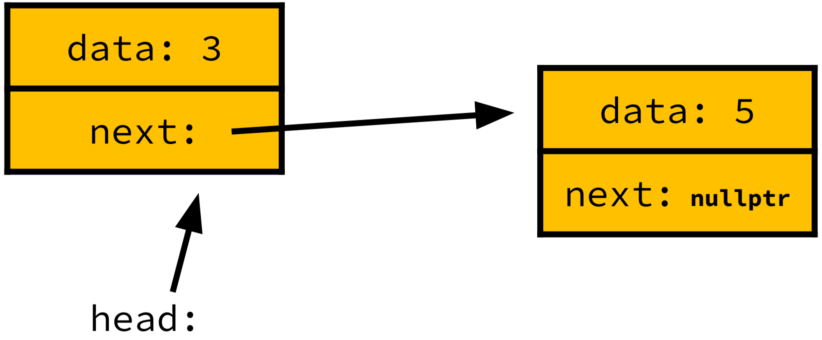


Passing Pointers by Reference

- When passed by reference, the callee function can change where the original pointer points

```
void prependTo(Node*& head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

int main() {
    Node* head = nullptr;
    prependTo(head, 5);
    prependTo(head, 3);
    return 0;
}
```



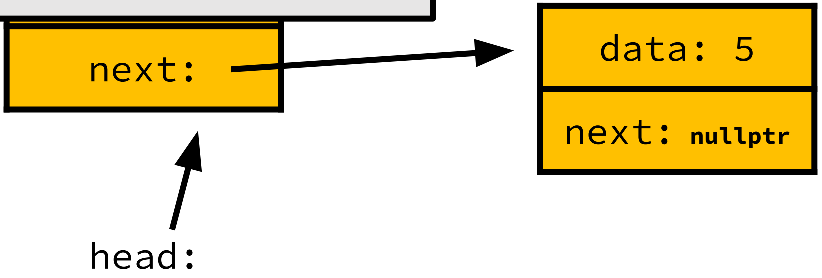
Passing Pointers by Reference

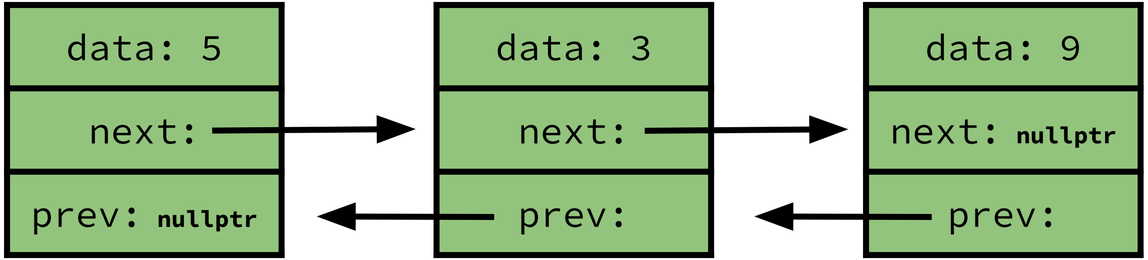
- When passed by reference, the callee function can change where the origina

```
void pre
     Node
     newN
     newN
     head
}

int main() {
     Node* head = nullptr;
     prependTo(head, 5);
     prependTo(head, 3);
     return 0;
}
```

When you want a helper function to modify the address a pointer points to, you should pass it by reference.

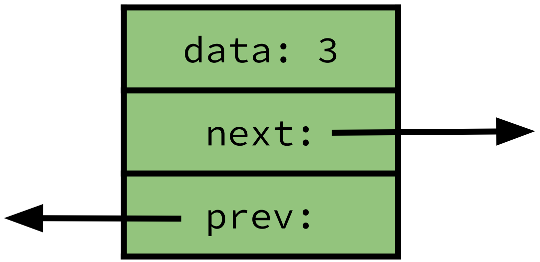




Doubly Linked Lists

Doubly Linked Lists

- Variation of linked lists that store a pointer to the next AND previous element in the list
- Allows us to traverse in both directions

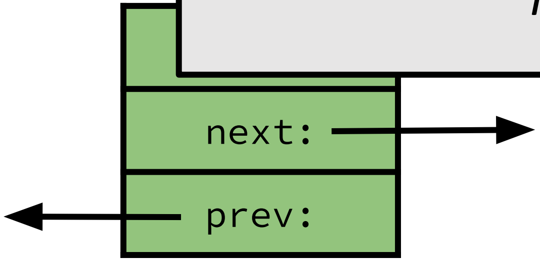



```
struct Node {
    int data;
    Node* next;
    Node* prev;
};
```

Doubly Linked Lists

- Variation of linked lists that store a pointer to the next AND previous el
- Allows us t

👥 *Discuss potential pros and cons of doubly linked lists compared to singly linked lists.*




```
{
    Node* next;
    Node* prev;
};
```

Recap

- Linked list recursion
  - We don’t traverse linked lists recursively!
- Big-O runtimes of linked list operations
- `createList` demo
- Pointers by reference
- Doubly linked lists

Thank you!