DATA STRUCTURES

Data

A collection of facts, concepts, figures, observations, occurrences or instructions in a formalized manner.

Information

The meaning that is currently assigned to data by means of the conventions applied to those data (i.e. processed data)

Record

Collection of related fields.

Data Type

Set of elements that share common set of properties used to solve a program.

Data Structure

It is the way of organizing, storing, and retrieving data and their relationship

DS Characteristics

1. It depicts the logical representation of data in computer memory.

DS Characteristics

2. It represents the logical relationship between the various data elements.

DS Characteristics

3. It helps in efficient manipulation of stored data elements.

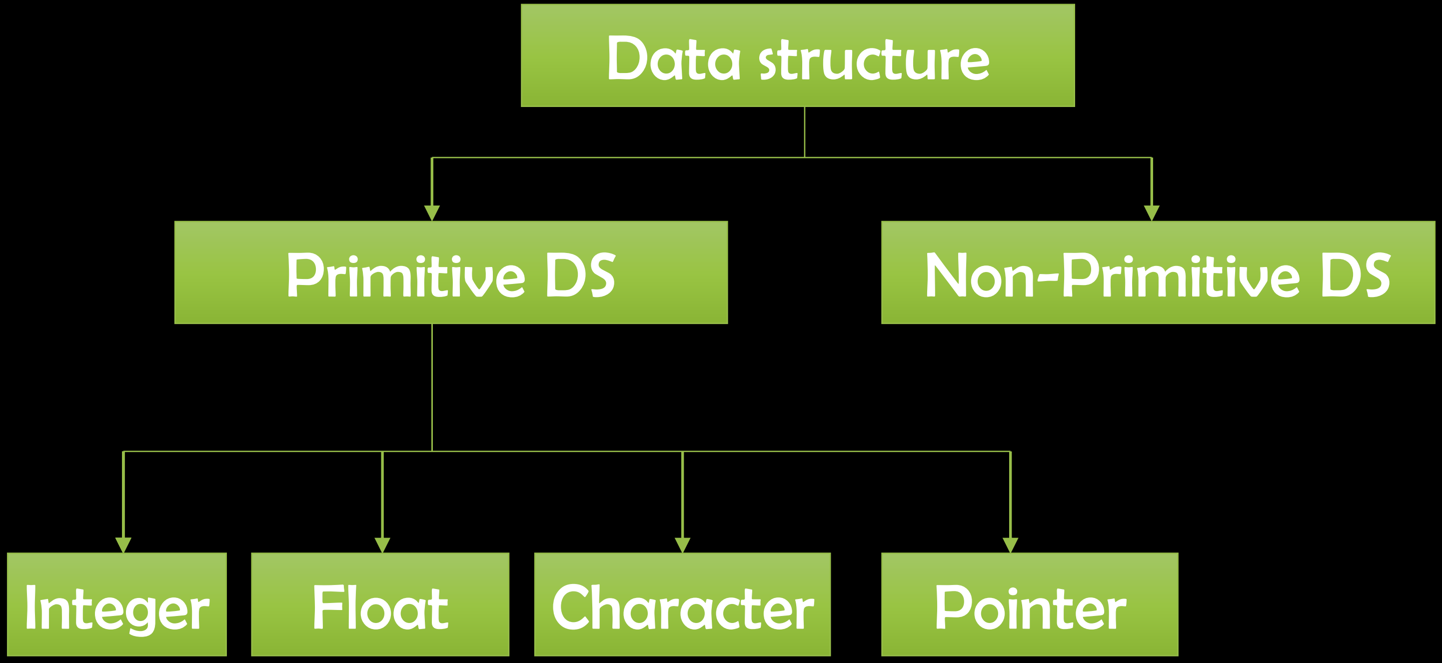
DS Characteristics

4. It allows the programs to process the data in an efficient manner.

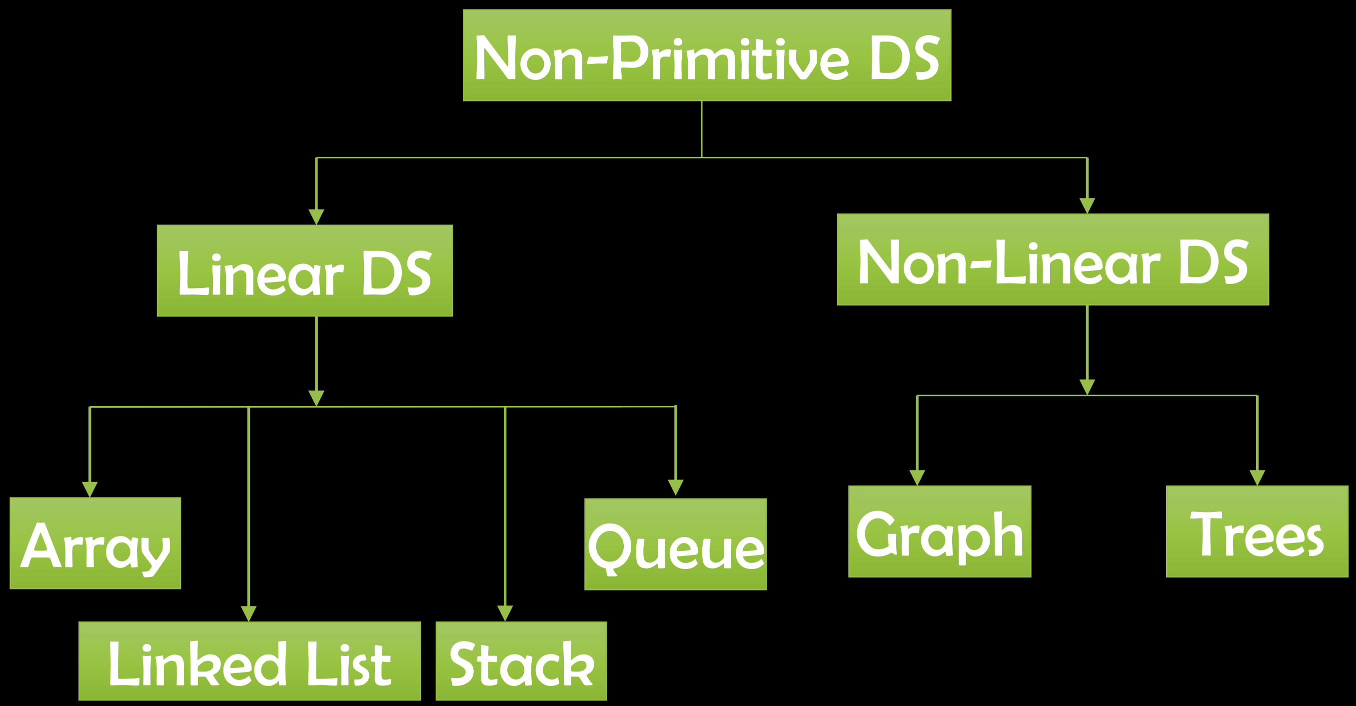
DS Operations

1. Traversal
2. Search
3. Insertion
4. Deletion

DS Classification



DS Classification



DS Applications

1. Compiler design
2. Operating system
3. Statistical analysis package
4. DBMS
5. Numerical analysis
6. Simulation
7. Artificial intelligence
8. Graphics

ABSTRACT DATA TYPES

ADTs

A mathematical model with a collection of operations defined on that model.

ADTs

An example can be a set of integers, together with the operations of union, intersection and set difference form

ADTs

Consists of data together with functions that operate on that data.

ADTs Advantages/Benefits

1. Modularity
2. Reuse
3. Code is easier to understand
4. Implementation of ADTs can be changed without requiring changes to the program that uses the ADTs.

LINEAR DATA STRUCTURES

LIST

Everyday List

- Groceries to be purchased
- Job to-do list
- List of assignments for a course
- Dean's list

List ADTs

A sequence of zero or more elements

$A_1, A_2, A_3, \ldots A_N$

- N: length of the list
- A_1: first element
- A_N: last element
- A_i: position i
- If N=0, then empty list
- Linearly ordered
  - A_i precedes A_{i+1}
  - A_i follows A_{i-1}

Operations

- printList: print the list
- makeEmpty: create an empty list
- find: locate the position of an object in a list
  - list: 34,12, 52, 16, 12
  - find(52) $\rightarrow$ 3
- insert: insert an object to a list
  - insert(x,3) $\rightarrow$ 34, 12, 52, x, 16, 12
- remove: delete an element from the list
  - remove(52) $\rightarrow$ 34, 12, x, 16, 12
- findKth: retrieve the element at a certain position

List ADT Implementation

1. Array based Implementation
2. Linked List based Implementation

Array

collection of related data

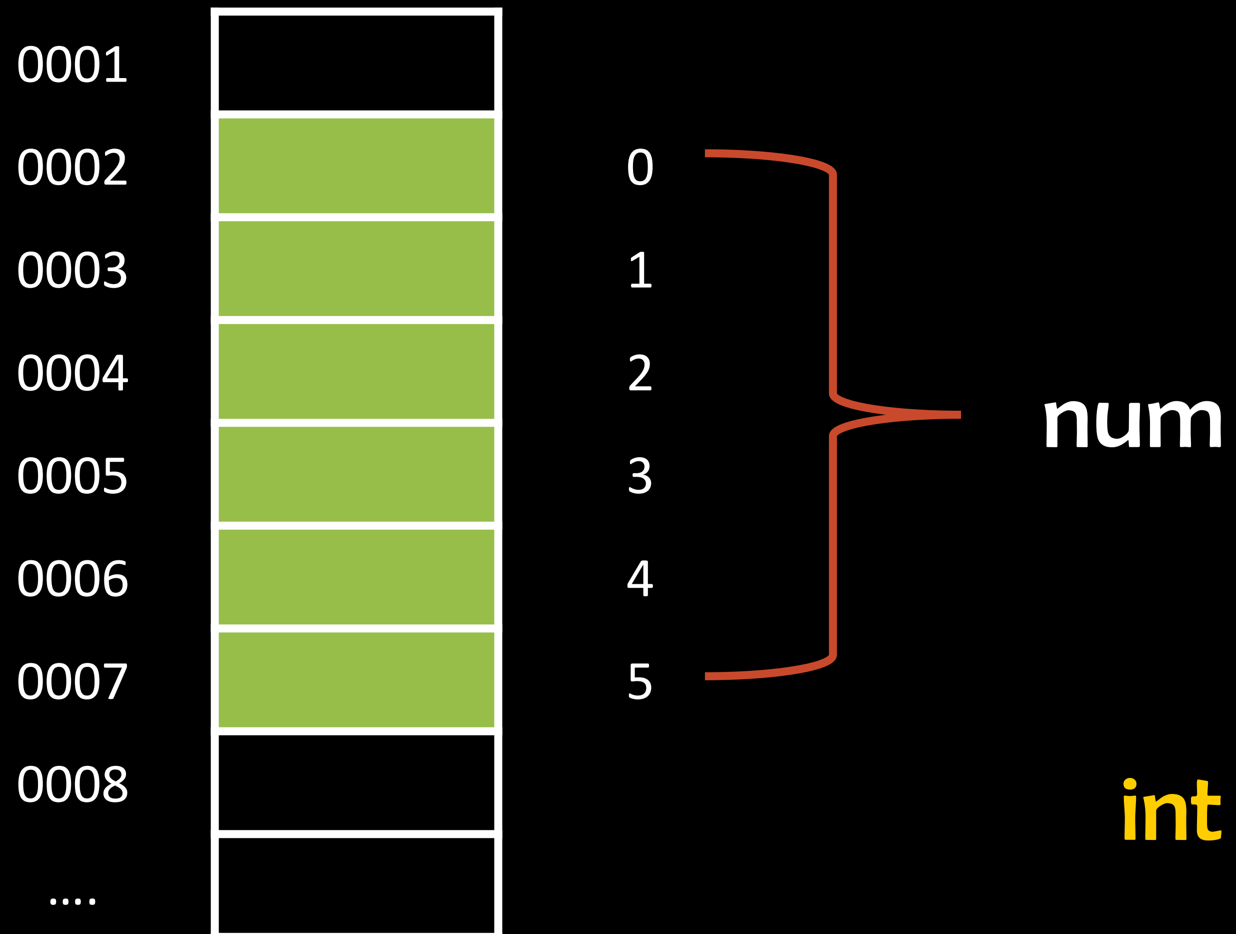
contents are of the same data type

Array

elements of the array are arranged

in a contiguous manner

```
int num[6];
```



int num[6];

```
insert(2, 0); //insert(value, position)
```

```
insert(2, 0);
```

2					
num[0]	num[1]	num[2]	num[3]	num[4]	num[5]

```
insert(4, 0);
```

num[0]	num[1]	num[2]	num[3]	num[4]	num[5]
2					

insert(4, 0);

```
insert(3, 1);
```

num[0]	num[1]	num[2]	num[3]	num[4]	num[5]
4	2				

```
insert(3, 1);
```

```
insert(8, 4);
```

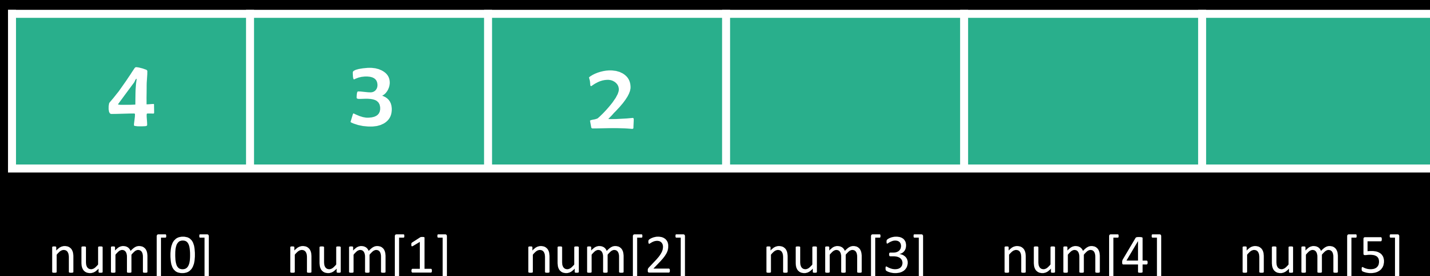
4	3	2			
num[0]	num[1]	num[2]	num[3]	num[4]	num[5]

insert(8, 4);

num[0]	num[1]	num[2]	num[3]	num[4]	num[5]
4	3	2			

operation not VALID!

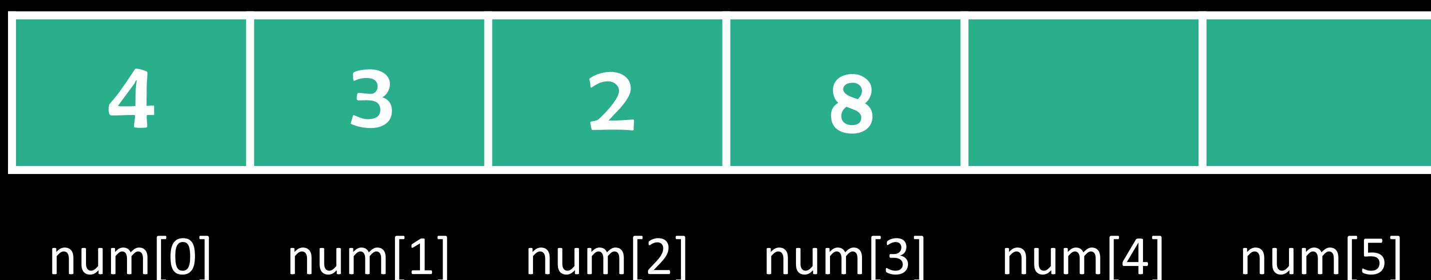
```
insert(8, 3);
```



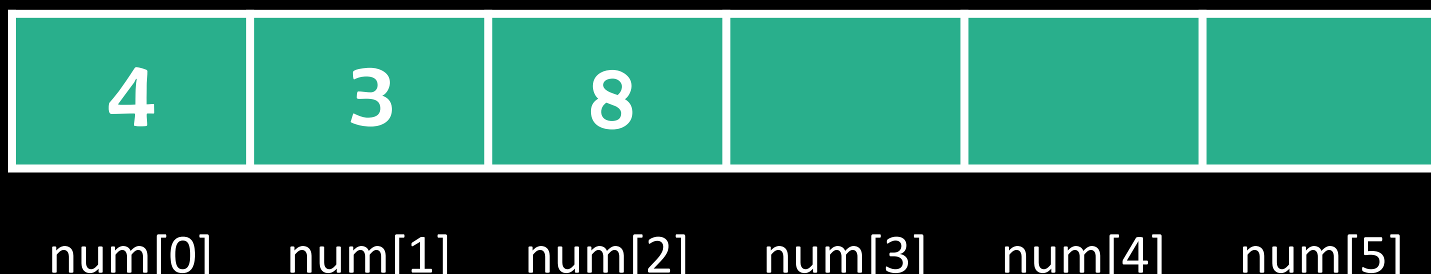
insert(8, 3)

num[0]	num[1]	num[2]	num[3]	num[4]	num[5]
4	3	2	8		

remove(2); //remove(position)



remove(2);



```
remove(8);
```

4	3	8			
num[0]	num[1]	num[2]	num[3]	num[4]	num[5]

remove(8);

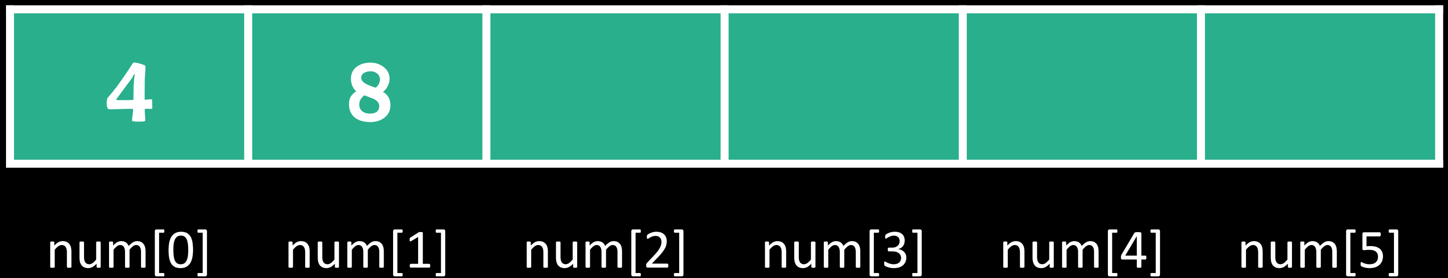
4	3	8			
num[0]	num[1]	num[2]	num[3]	num[4]	num[5]

operation not VALID!

remove(1);

4	3	8			
num[0]	num[1]	num[2]	num[3]	num[4]	num[5]

remove(1);



search(1);

4	8				
num[0]	num[1]	num[2]	num[3]	num[4]	num[5]

search(1);

value not FOUND!

search(4);

4	8				
num[0]	num[1]	num[2]	num[3]	num[4]	num[5]

search(4);

4	8				
num[0]	num[1]	num[2]	num[3]	num[4]	num[5]

value FOUND at index 0!

- printList and find: **O(n)**
- findKth: **O(1)**
- insert and delete: **O(n)**
  - e.g. insert at position 0 (making a new element)
    requires first pushing the entire array down one
    spot to make room
  - e.g. delete at position 0
    requires shifting all the elements in the list up one
  - On average, half of the lists needs to be moved for either operation

Array Implementation

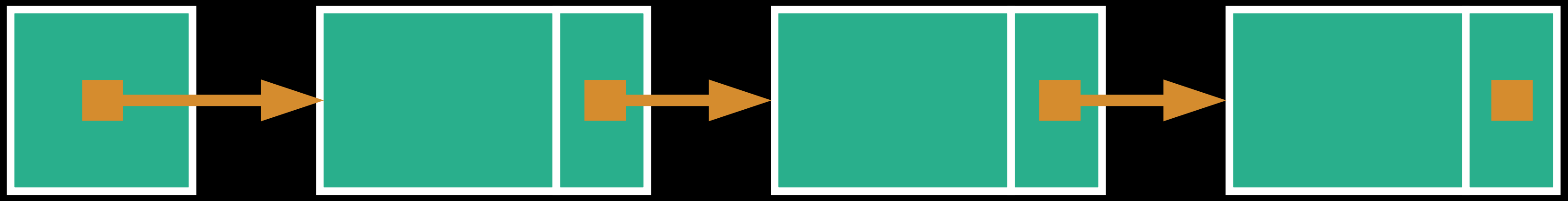
- Easy to implement
- Fast operation: findKth
- Size is fixed (compile time or run-time)
- Expensive operations: insert and delete

Linked List

a data structure that consists of
dynamic variables linked together
to form a chain-like structure

Linked List

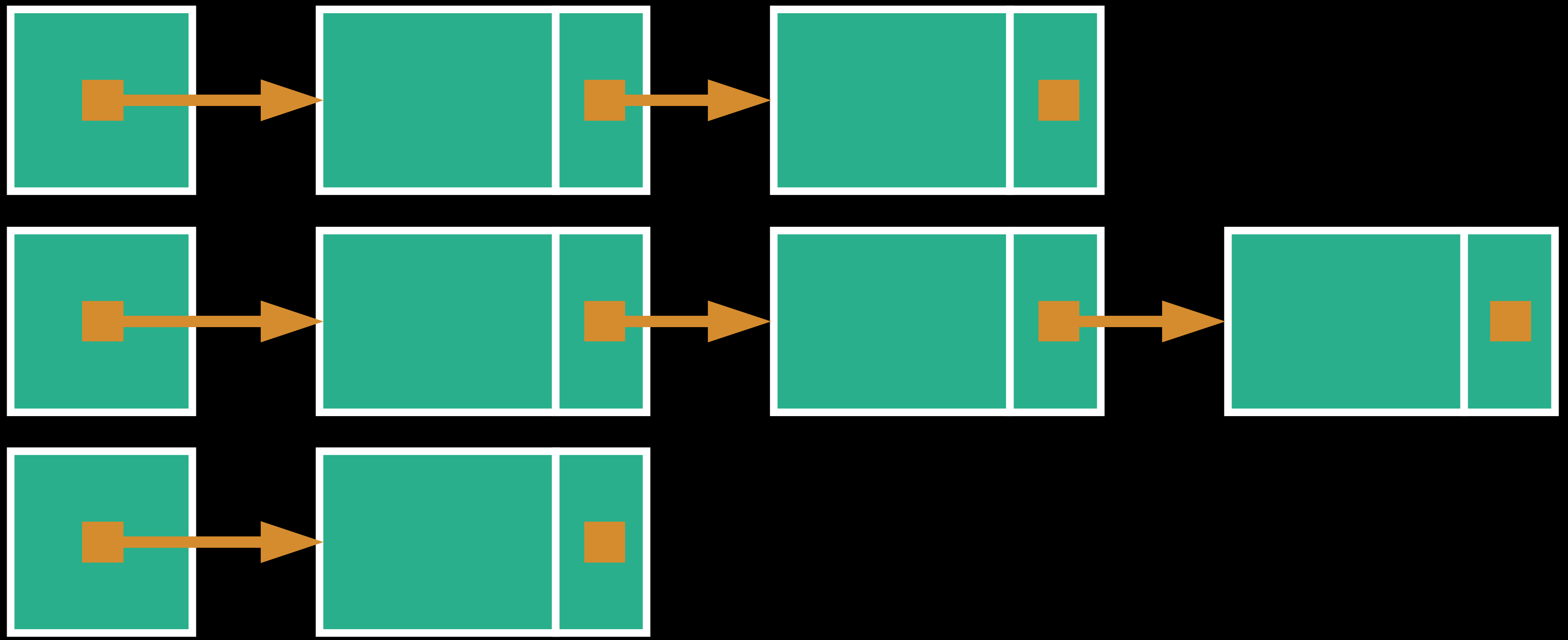
a data structure that consists of dynamic variables linked together to form a chain-like structure



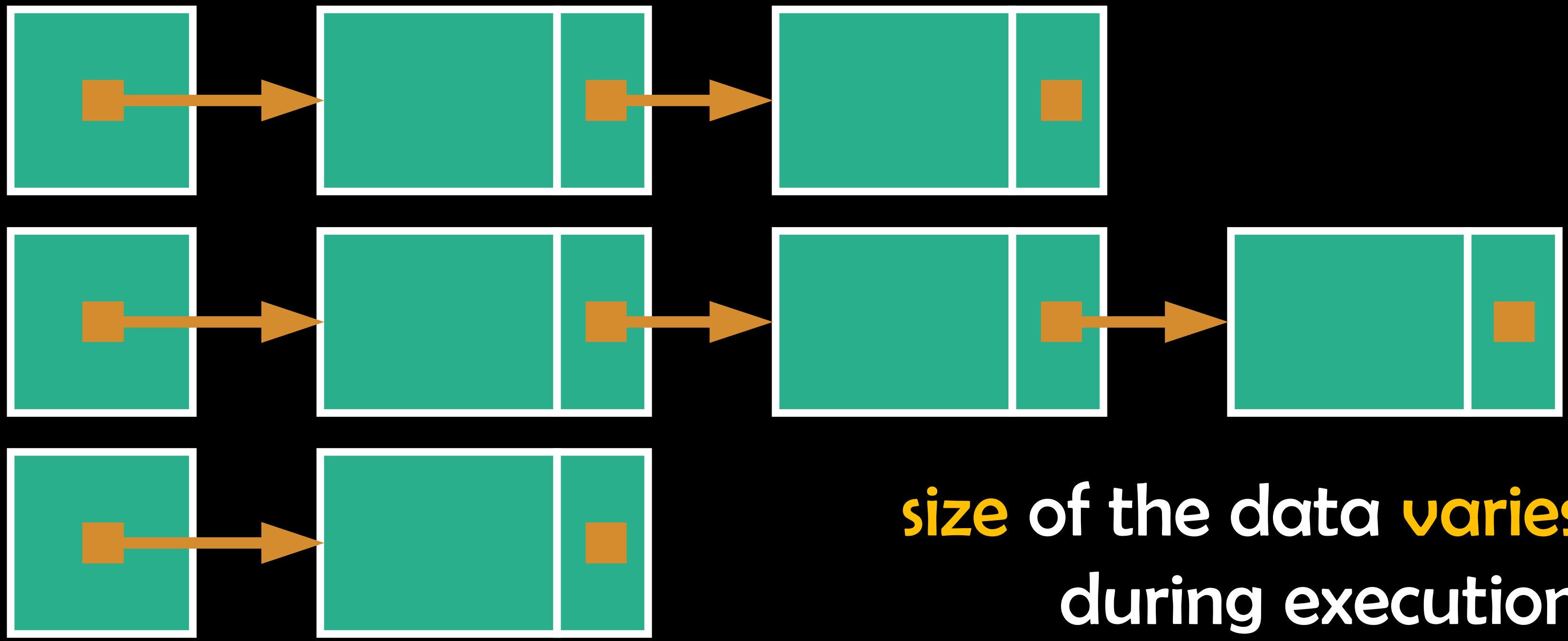
Linked List

an alternative to arrays
during execution, linked lists
can either grow or shrink
following the user's needs

Linked List

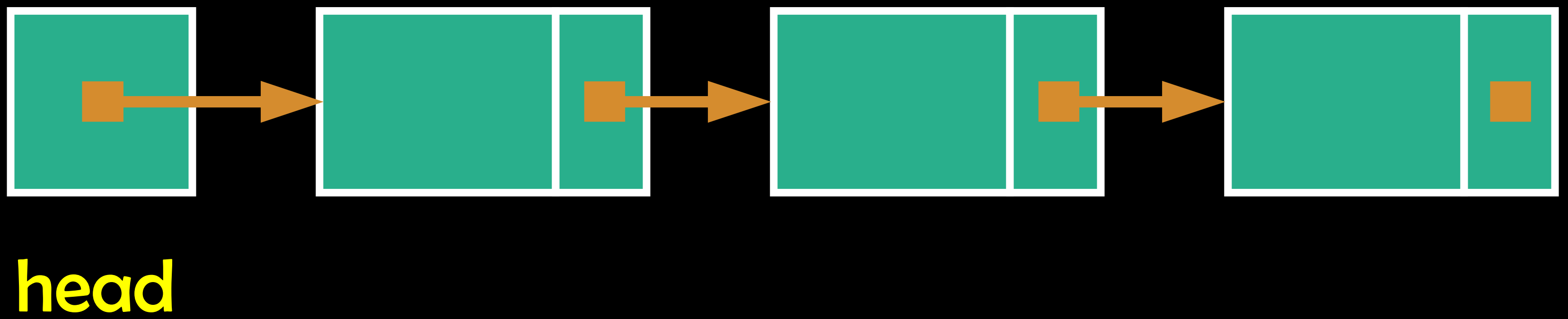


Linked List

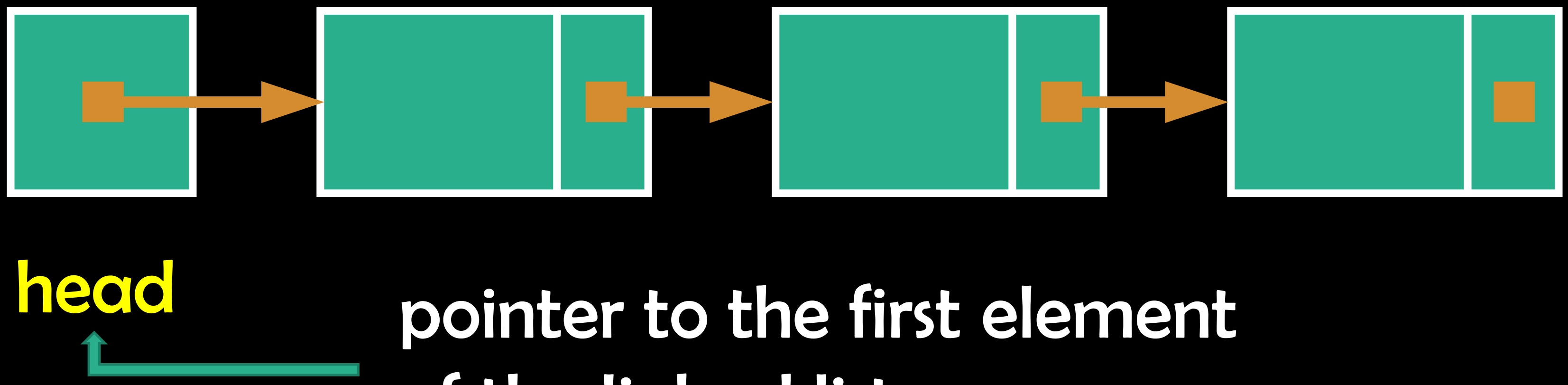


size of the data varies
during execution

Linked List

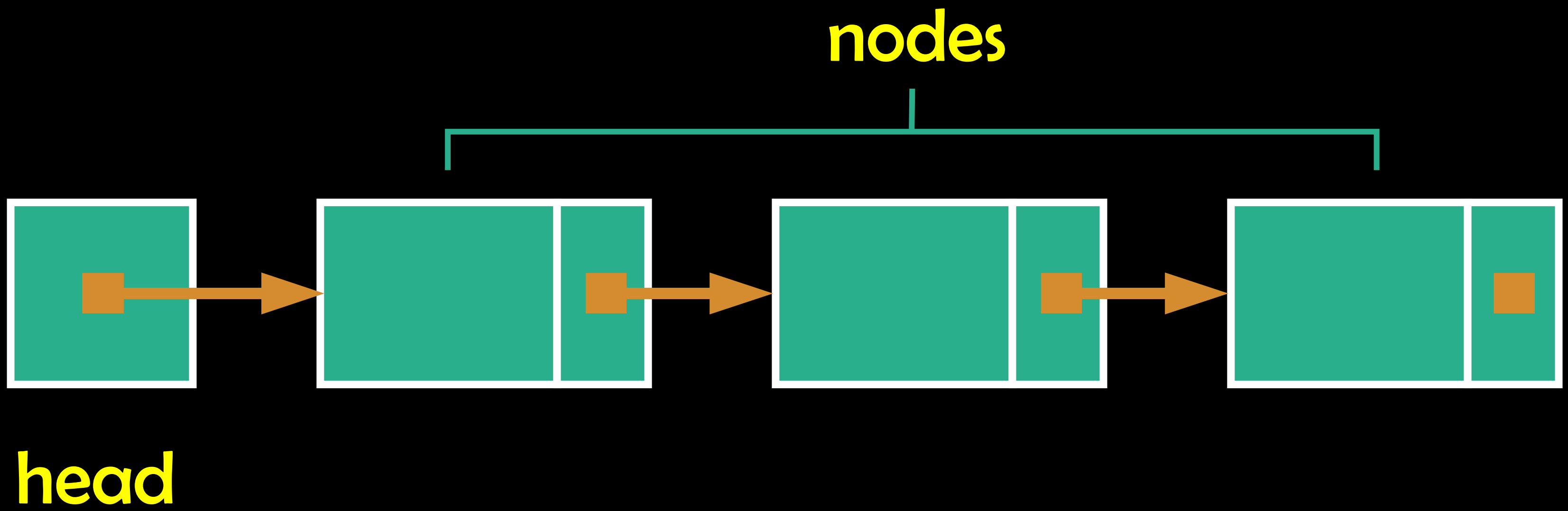


Linked List

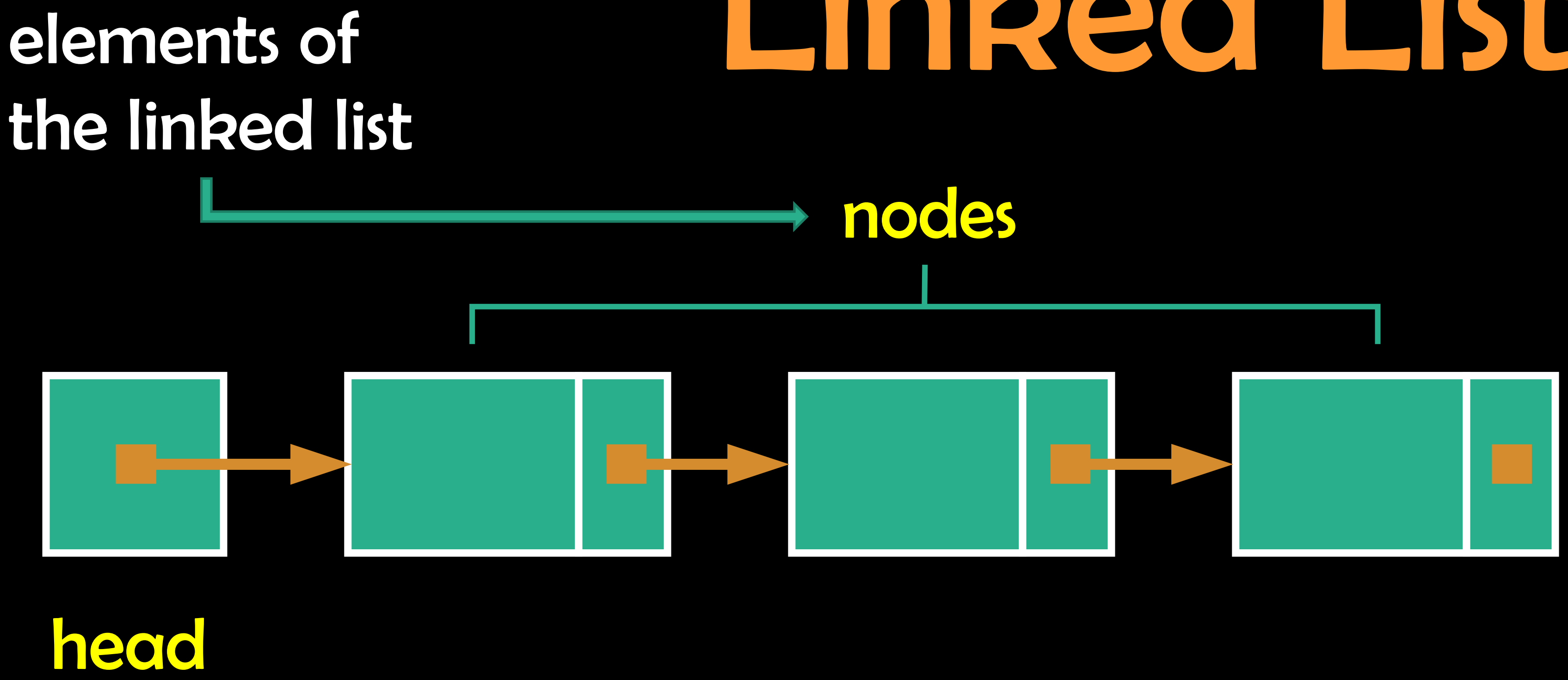


pointer to the first element of the linked list

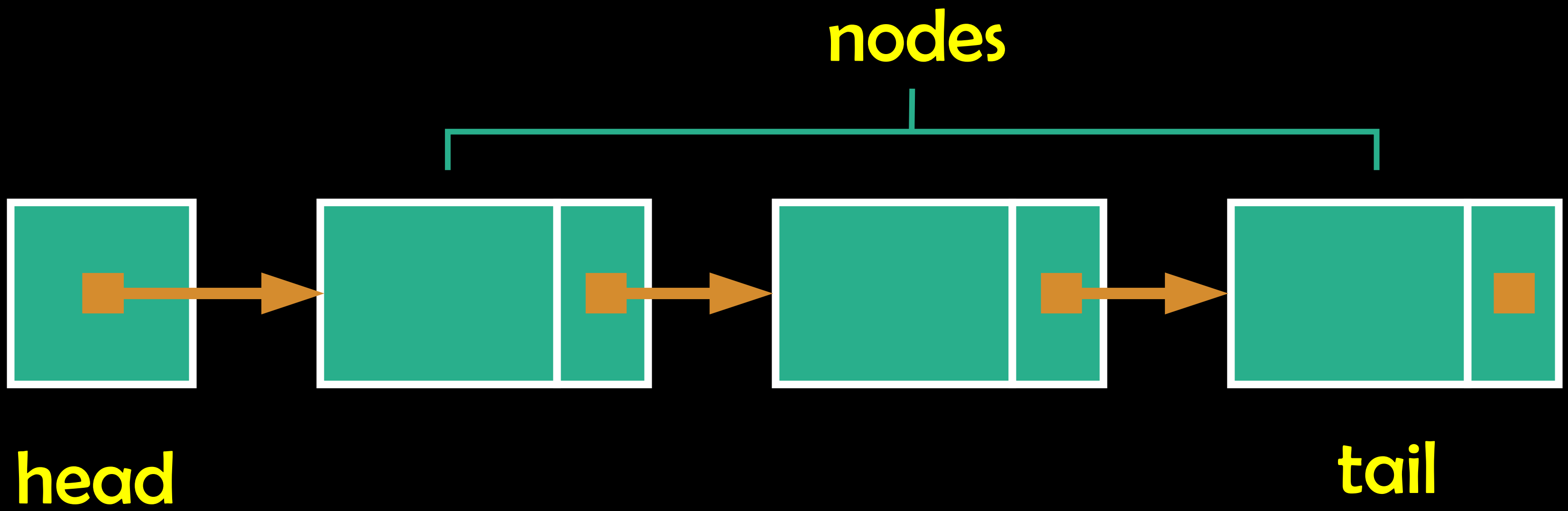
Linked List



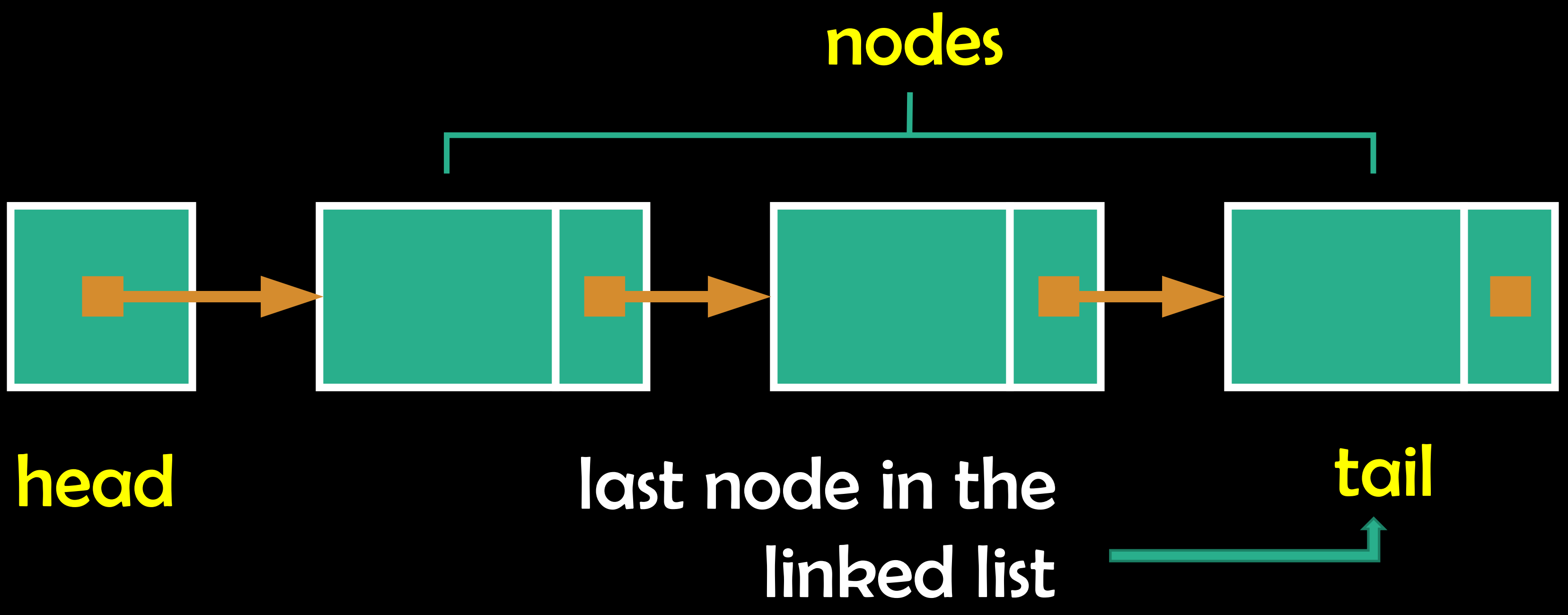
Linked List



Linked List



Linked List



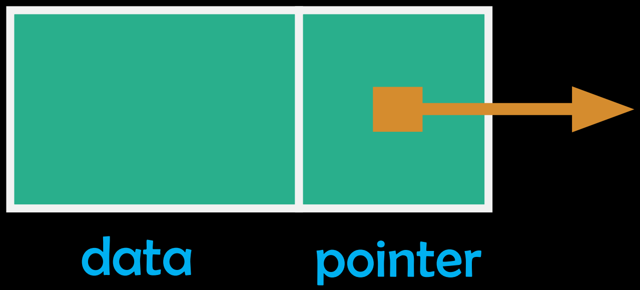
Linked List

Each node contains at least

- A piece of data (any type)
- Pointer to the next node in the list

Last node points to NULL

node



- printList and find: **O(n)**
- findKth: **O(n)**
- insert and delete: **O(1)**
  - e.g. insert at position 0 (making a new element)
    Insert does not require moving the other elements
  - e.g. delete at position 0
    requires no shifting of elements
  - Insertion and deletion becomes easier, but finding the kth element moves from O(1) to O(n)

LL Implementation

- Size can grow or shrink
- Easier to do: insert and delete
- Difficult to implement (?)
- findKth no longer faster

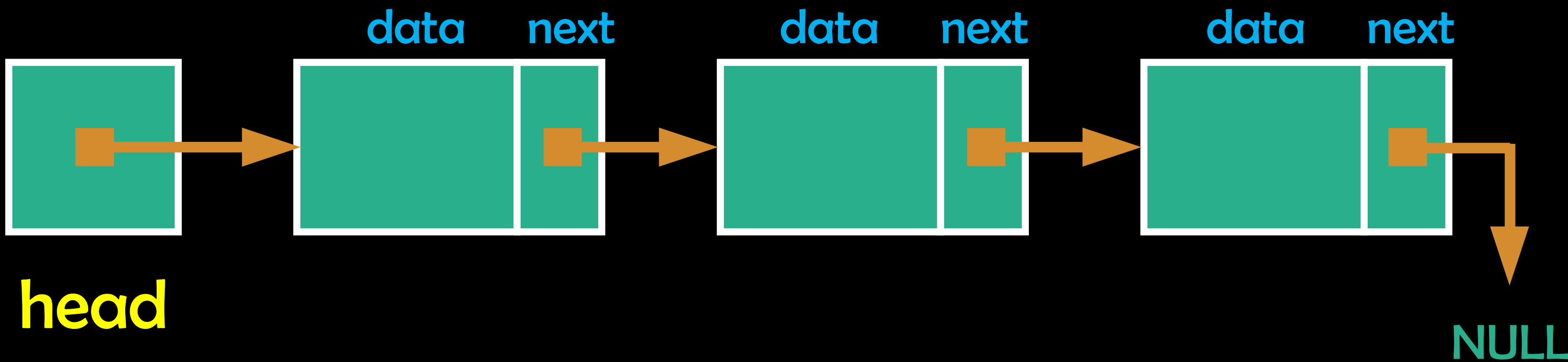
Kinds of Linked Lists

- Singly Linked Lists
- Doubly Linked Lists
- Circular Linked Lists

(Singly) Linked List

item navigation is forward only.

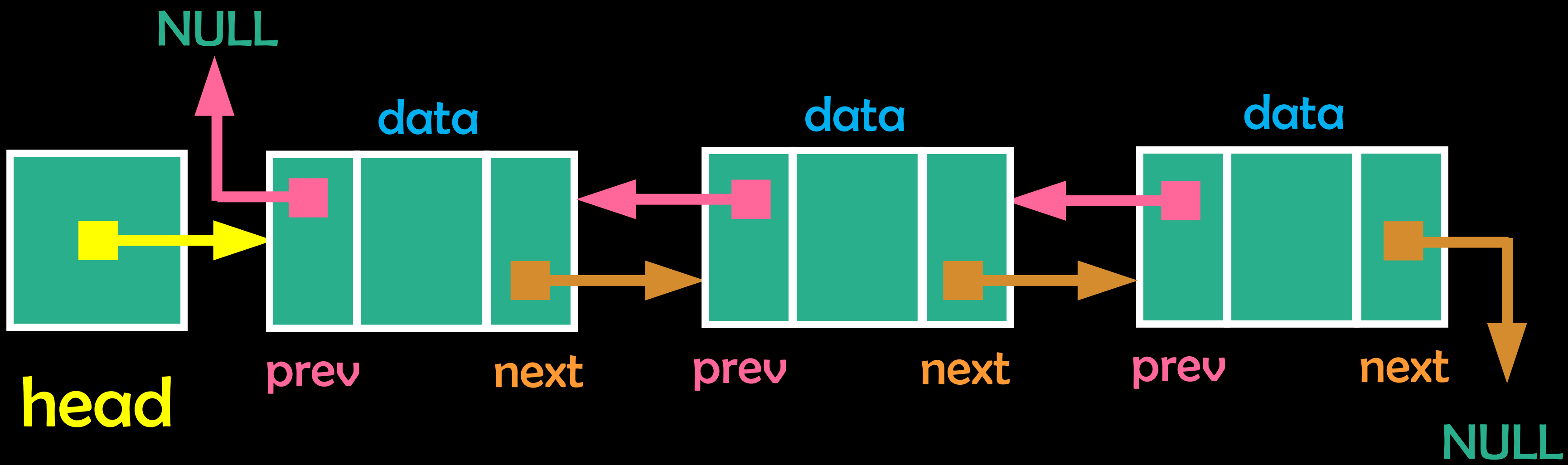
(Singly) Linked List



Doubly Linked List

item can be navigated forward and backward.

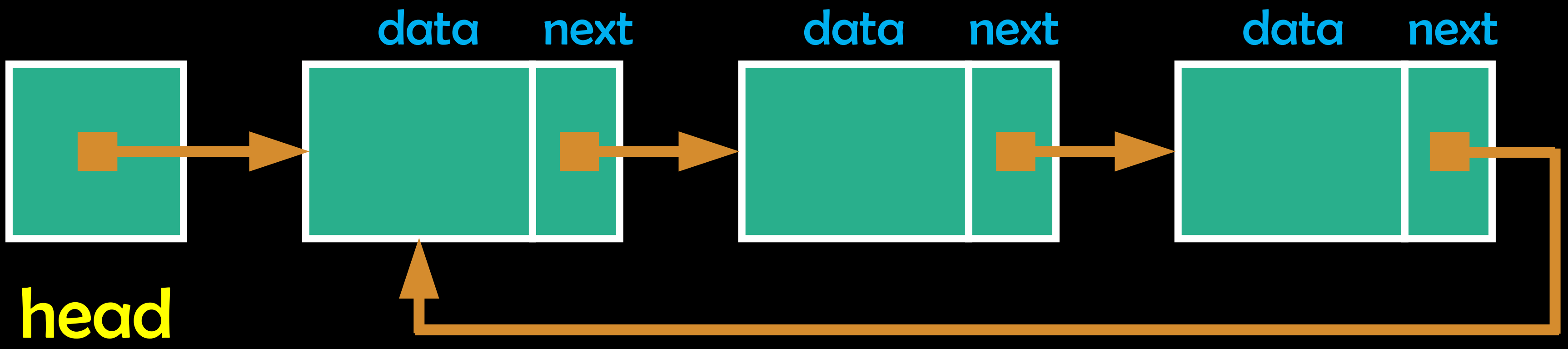
Doubly Linked List



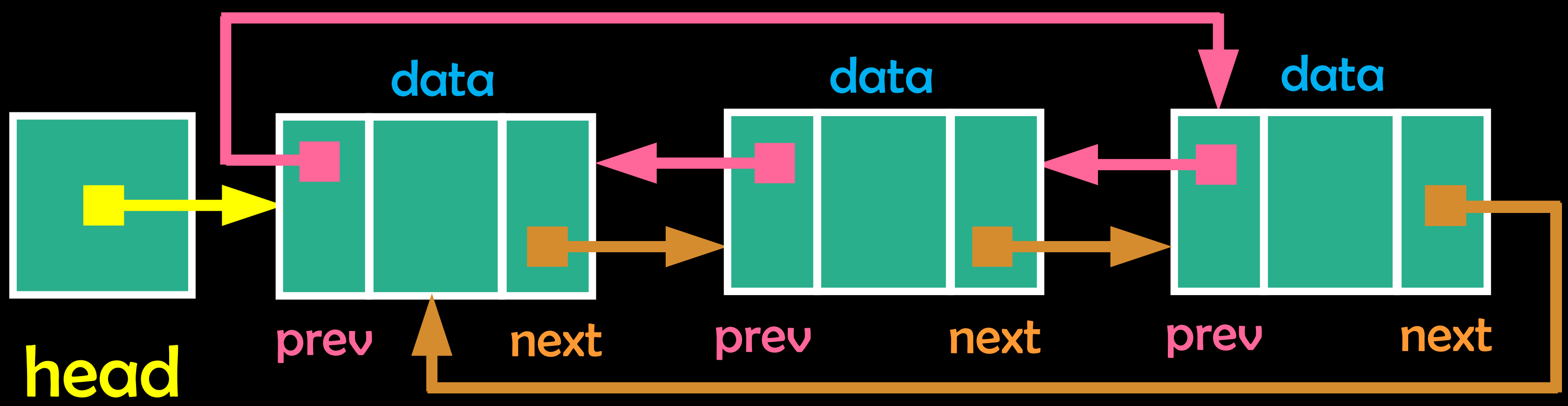
Circular Linked List

Last item contains link of the first element as next and the first element has a link to the last element as previous.

Singly Circular Linked List



Doubly Circular Linked List



Operations

Insert (add a node)

Delete (delete a node)

Search (search the list)

View (print the contents of the list)

Insert

insert values to a linked list

Insert

insert values to a linked list

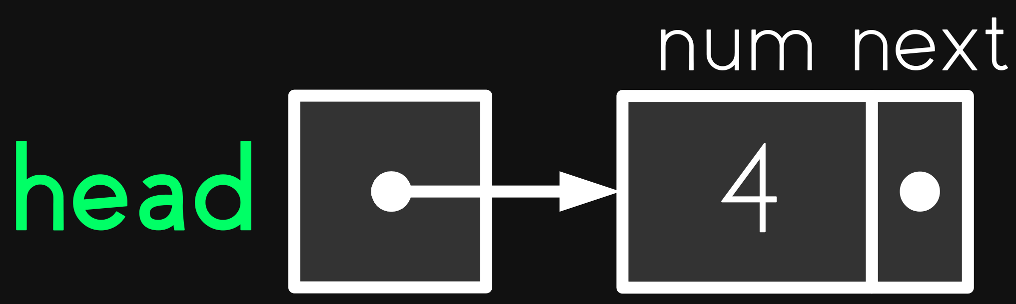
has three(3) cases:

- insert at head
- insert at middle
- insert at tail

Insert at Head

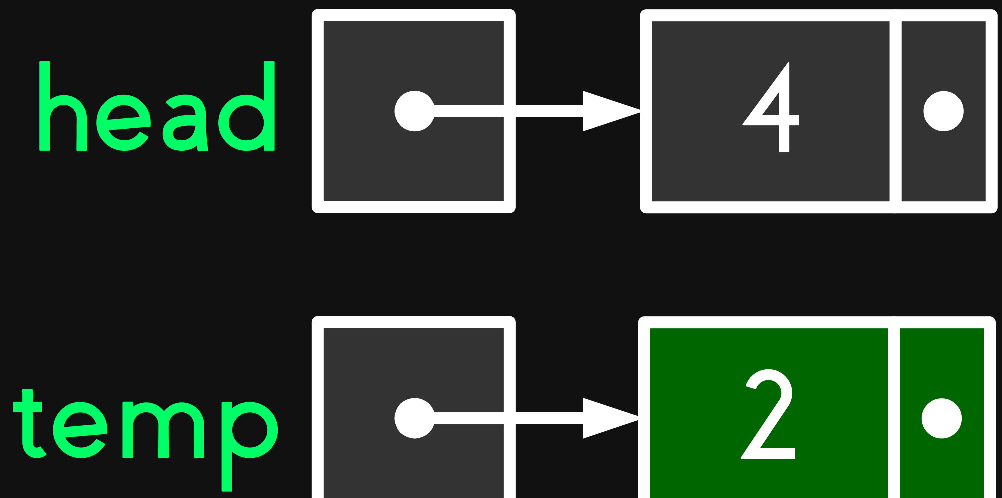
insert a node
at the beginning
of the list

Insert at Head



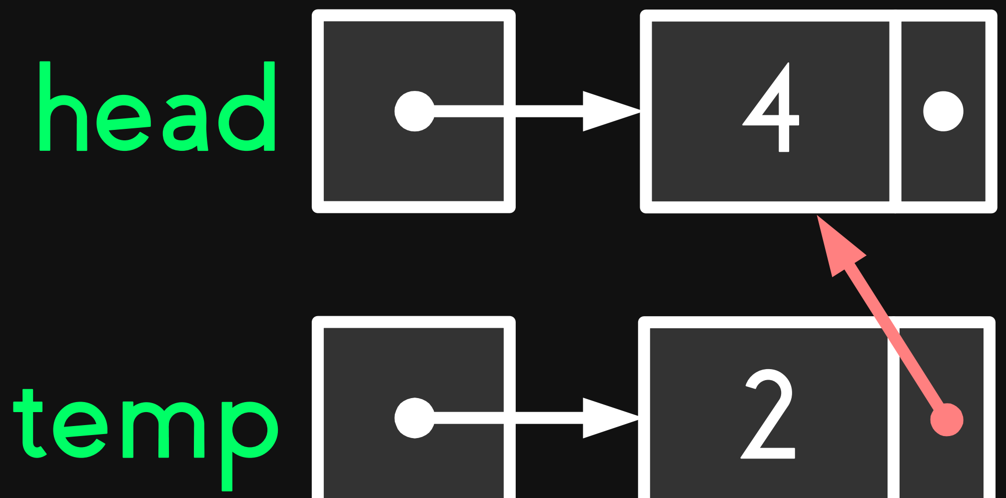
insert 2 at the start of the list

Insert at Head



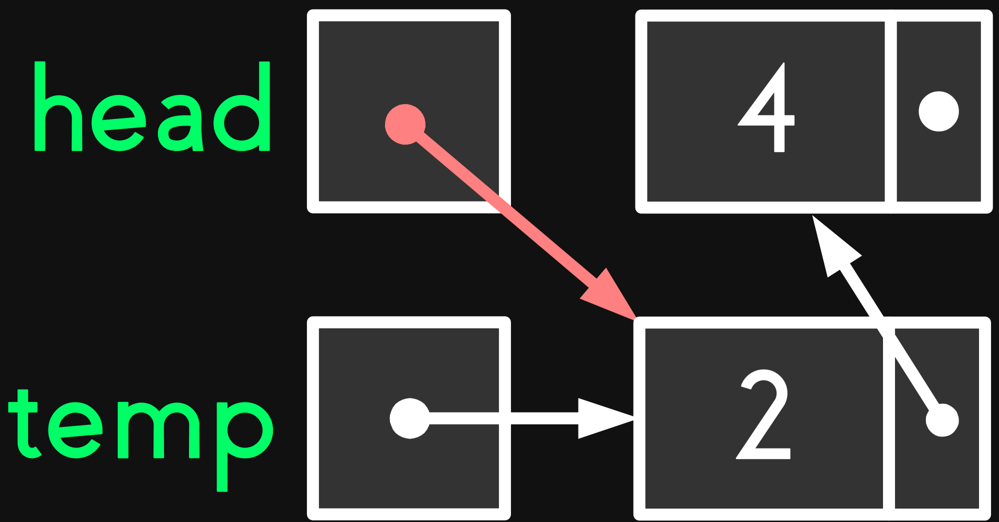
create a new node for 2.
give it to a new pointer (temp).

Insert at Head



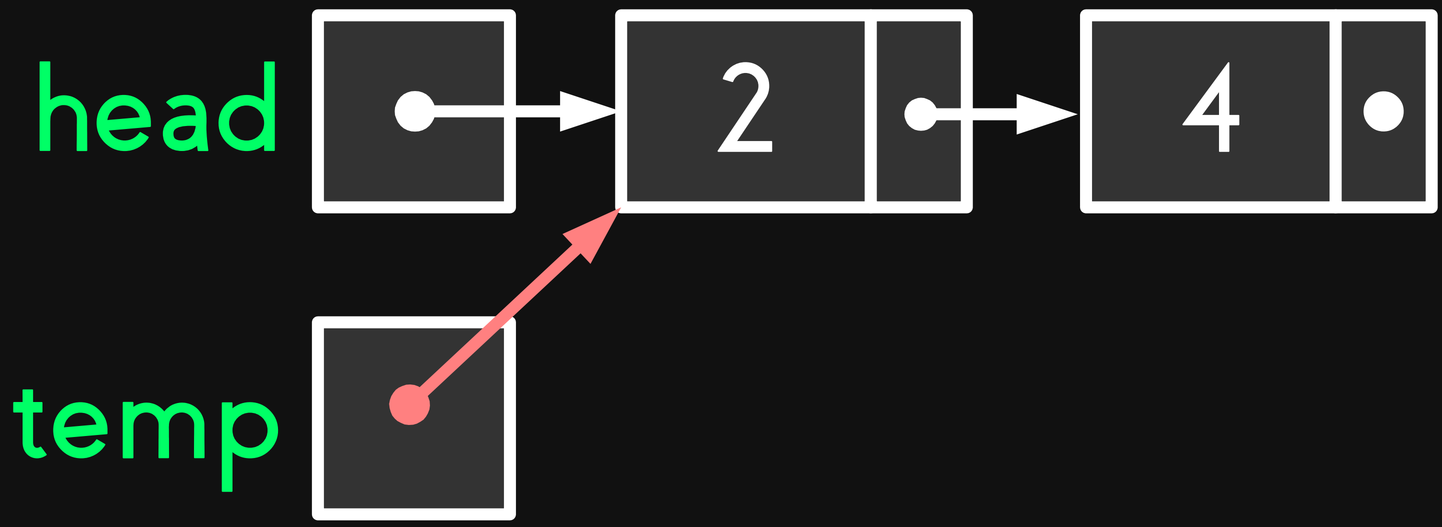
make the next pointer of the new node (2) point to the current head (4).

Insert at Head



make the head pointer point
to the new node (2).

Insert at Head

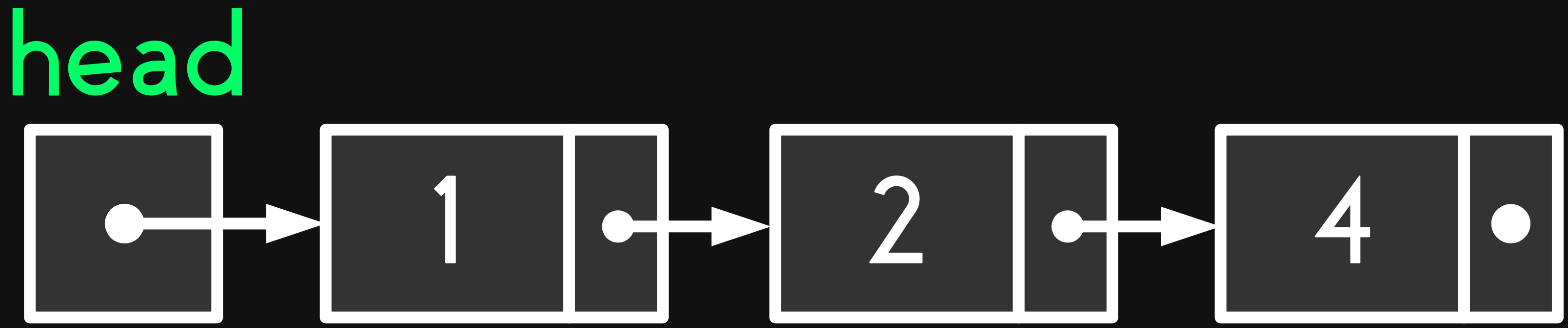


rearrangement of the
previous diagram.

Insert at Middle

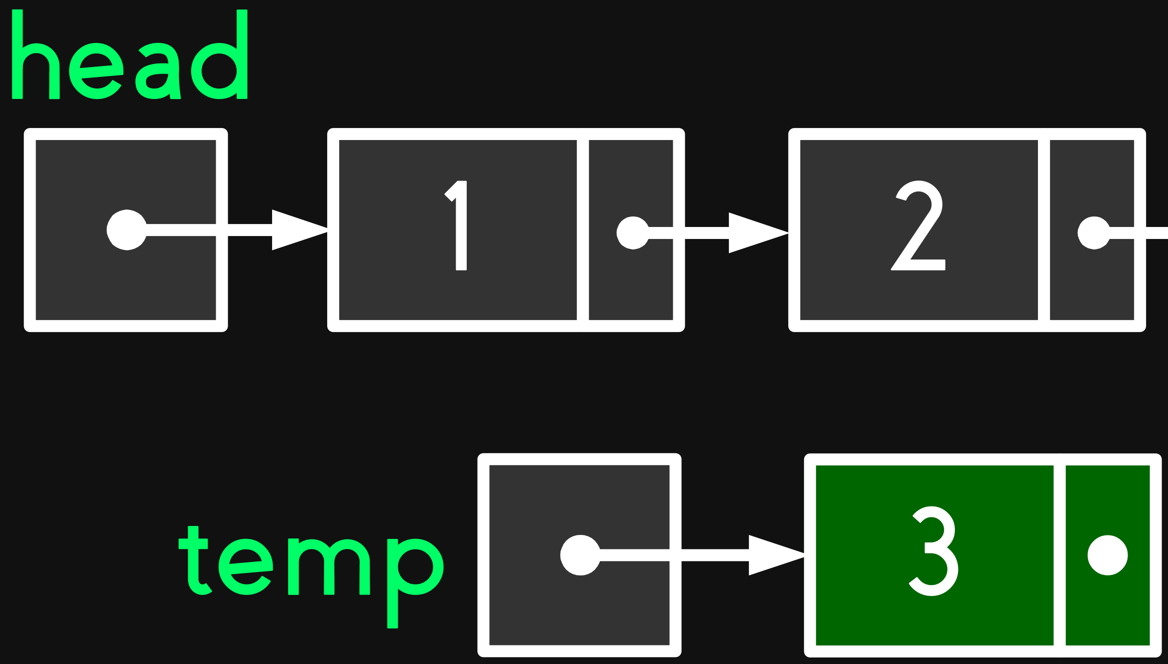
insert a node
between two nodes
of the list

Insert at Middle



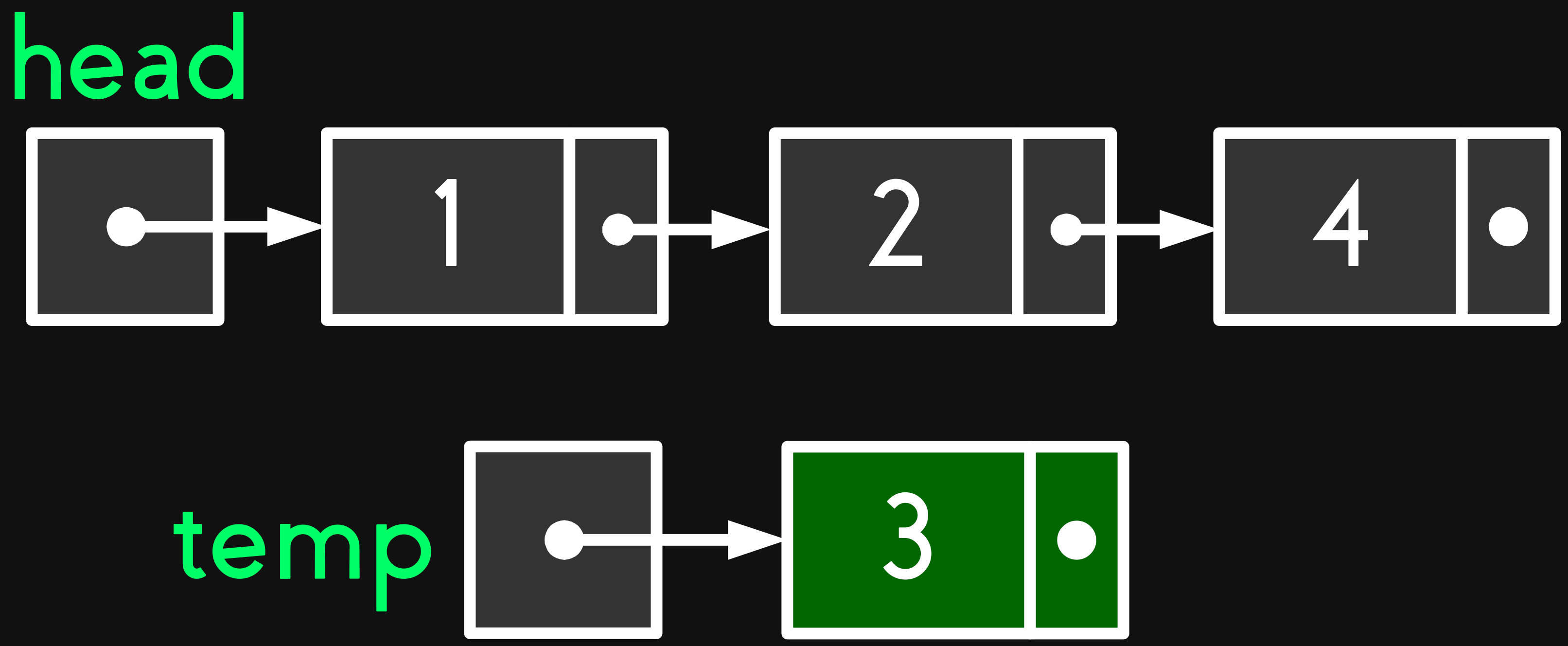
insert 3 in the middle
of the linked list.

Insert at Middle



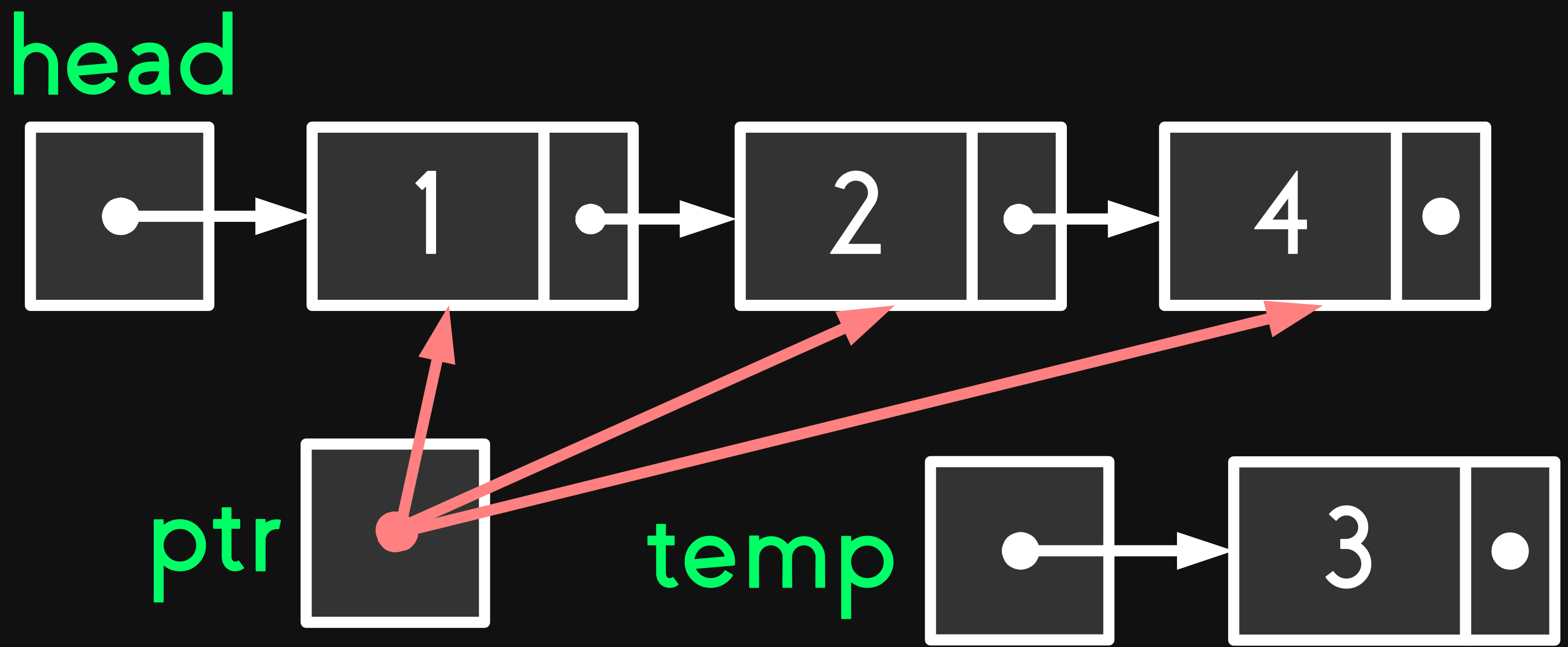
create a new node for 3.
give it to a new pointer (temp).

Insert at Middle



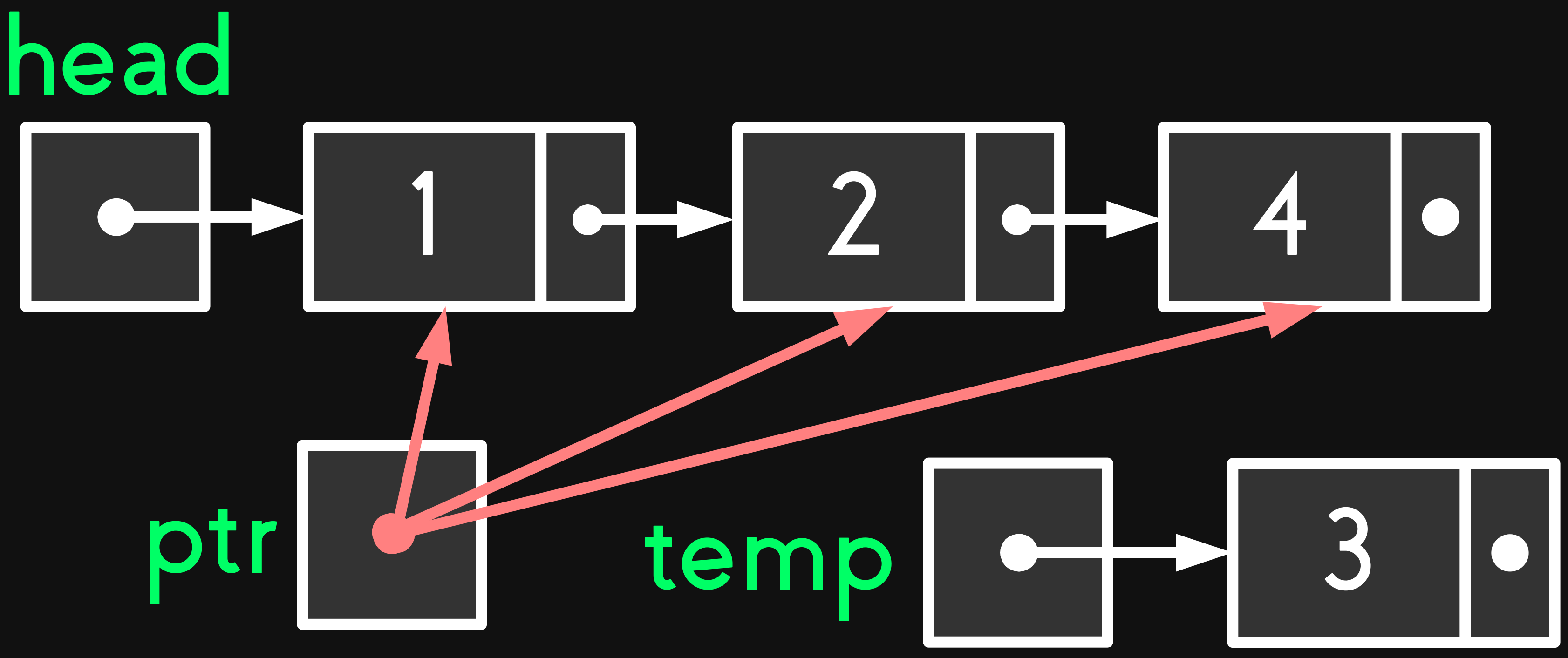
find the position where the new node is to be inserted.

Insert at Middle



select the node **before** the position where the new node will be inserted

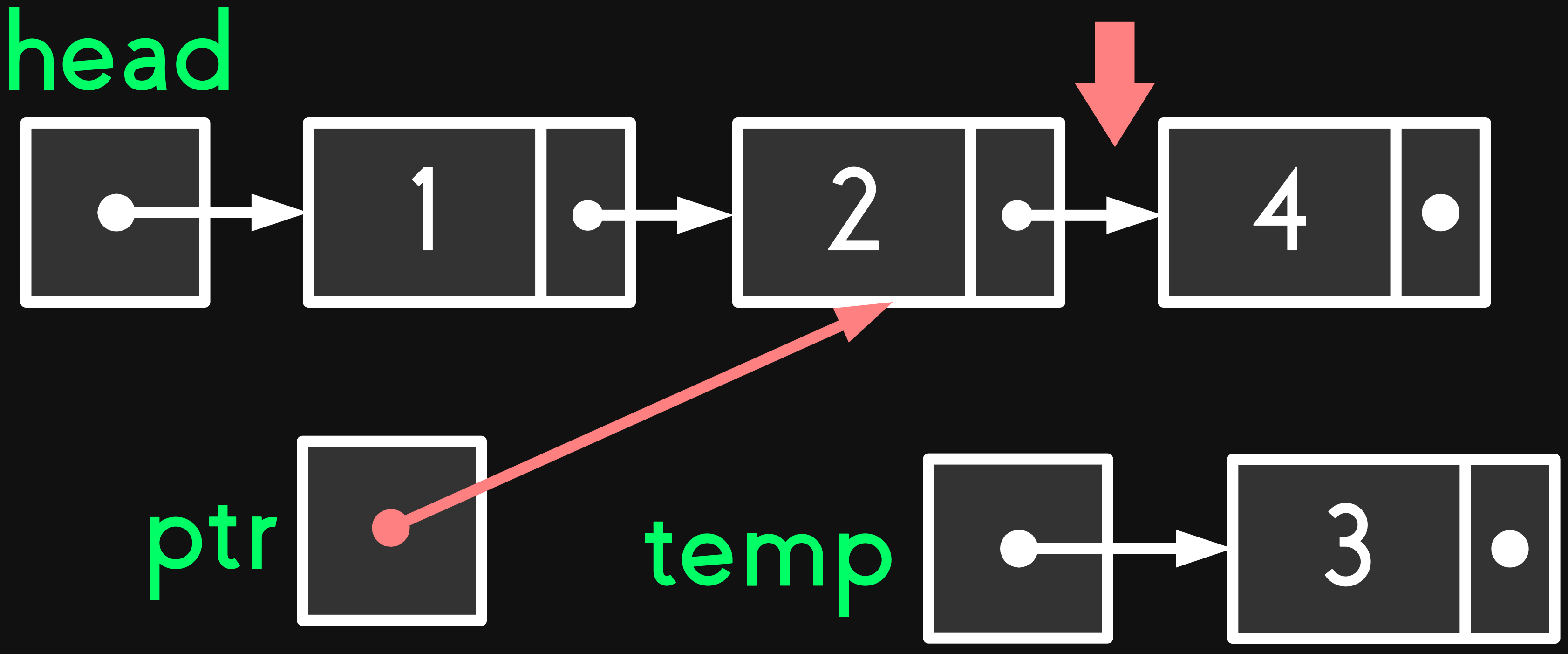
Insert at Middle



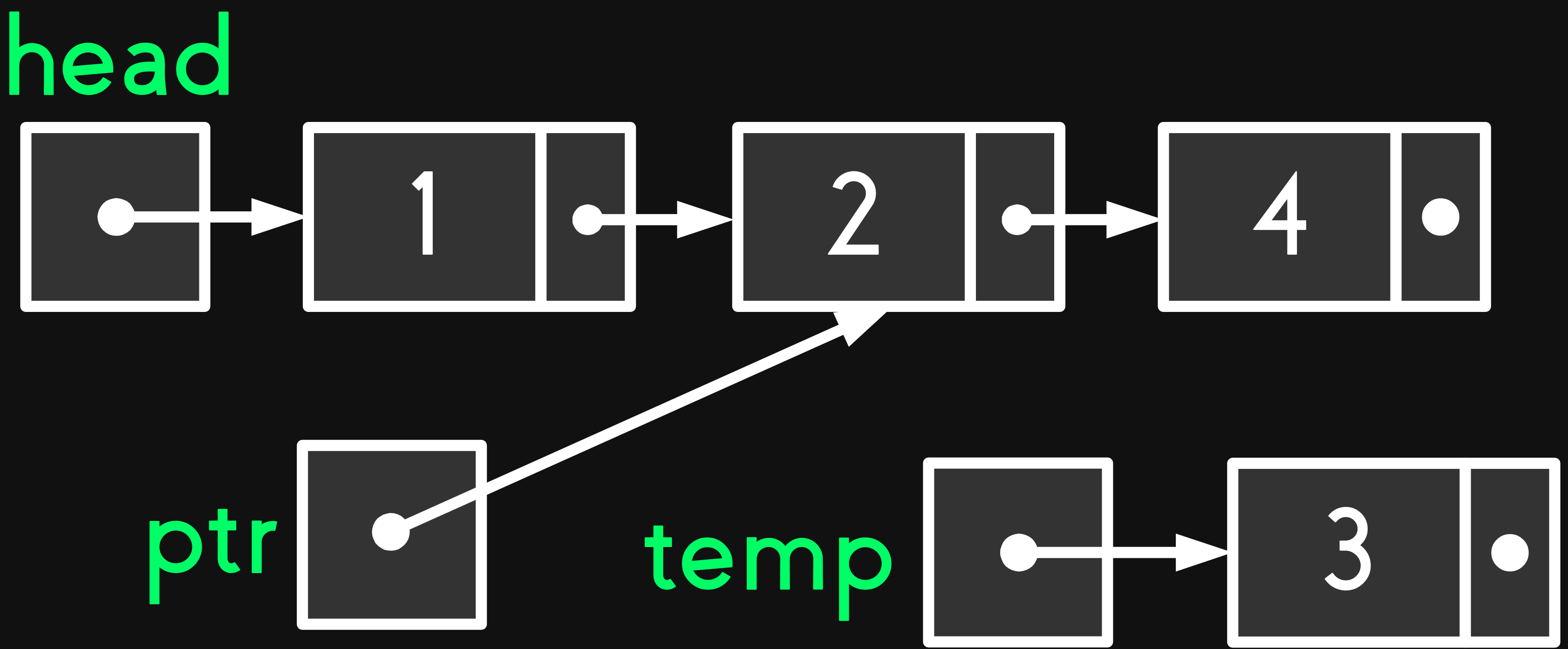
let's say we want to insert the node containing 3 in between the nodes containing 2 and 4.

Where should ptr point?

Insert at Middle

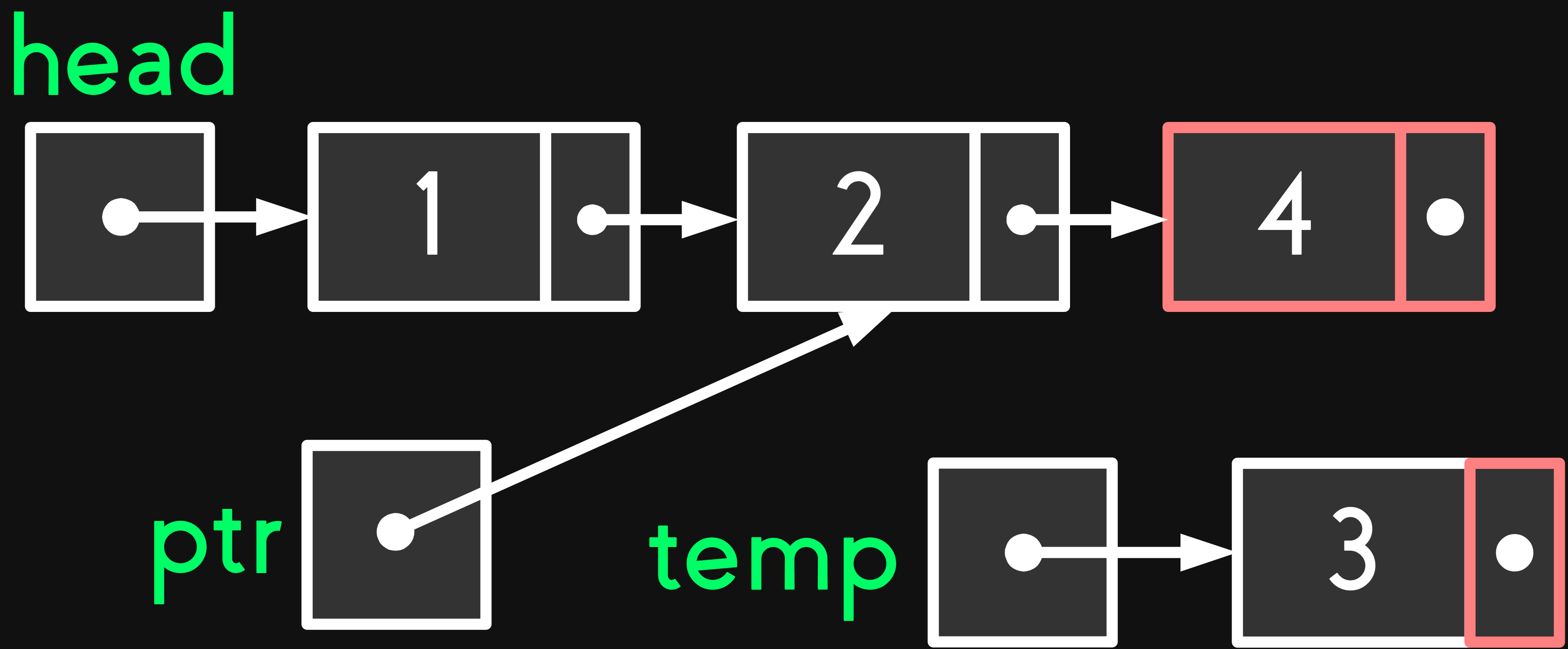


Insert at Middle



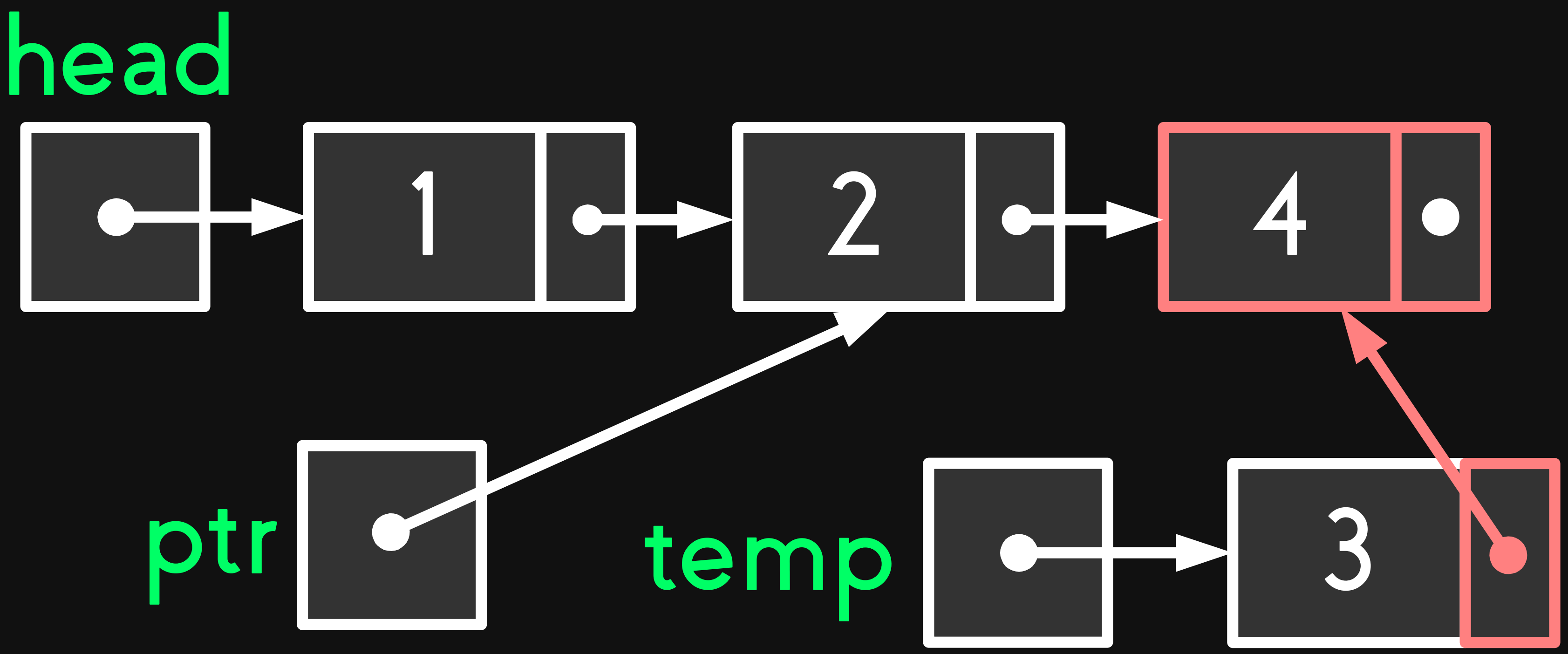
point the `next` pointer of the new node to the node being pointed by the `next` of the node being pointed by `ptr`

Insert at Middle



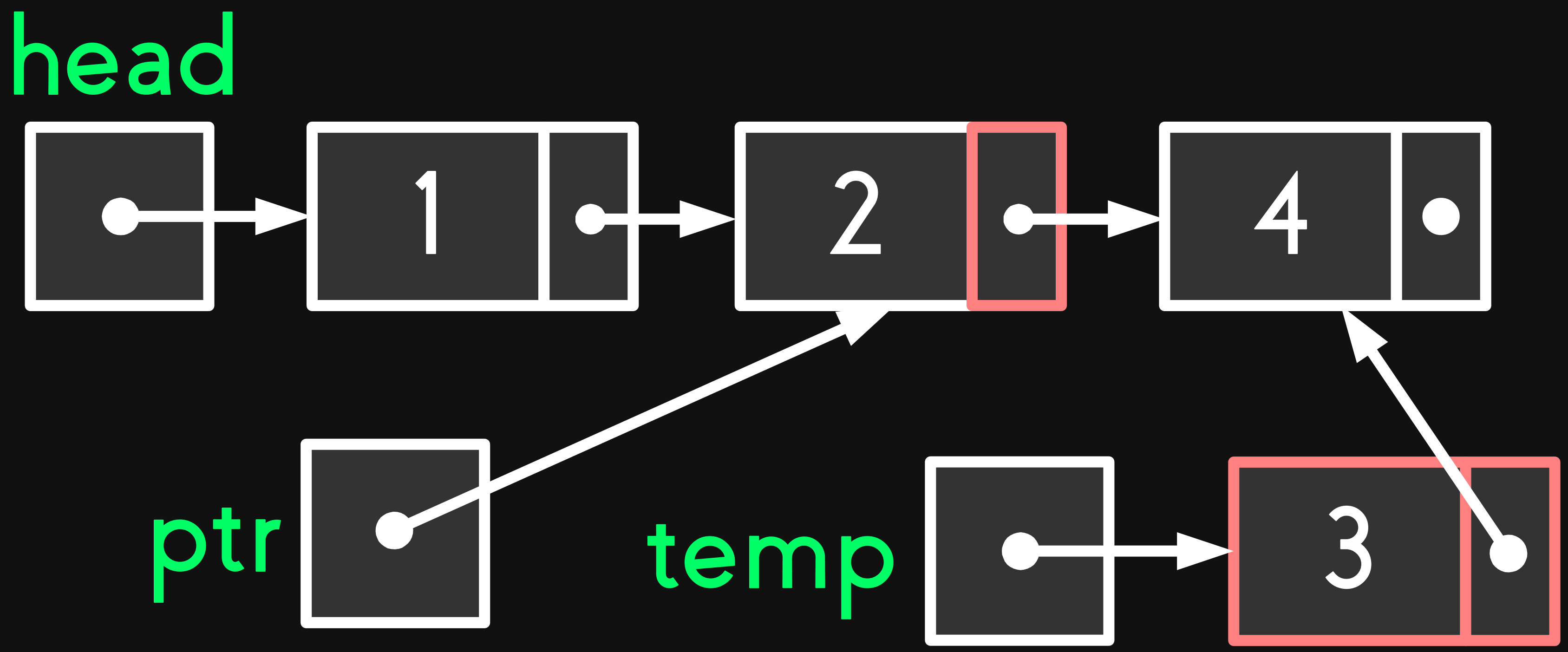
point the **`next`** pointer of the new node to the node being pointed by the **`next`** of the node being pointed by **`ptr`**

Insert at Middle



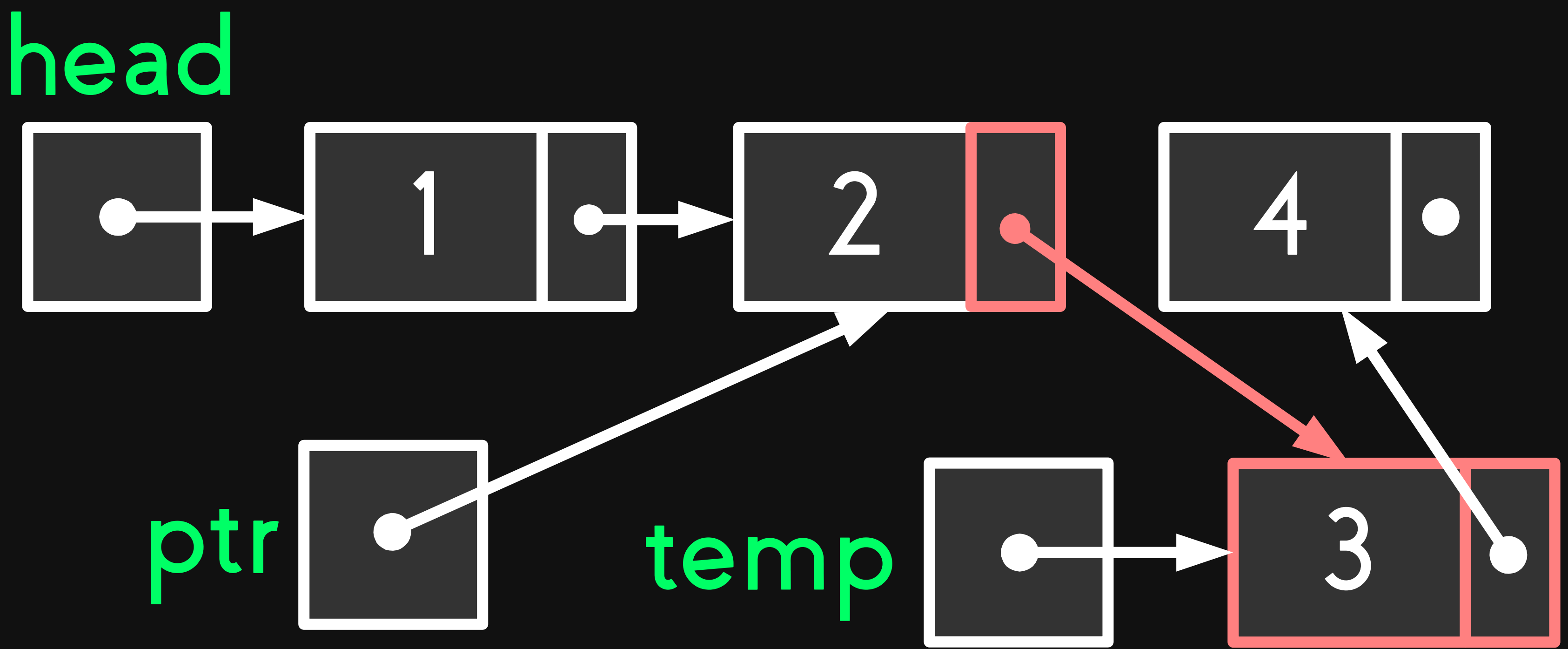
point the `next` pointer of the new node to the node being pointed by the `next` of the node being pointed by `ptr`

Insert at Middle



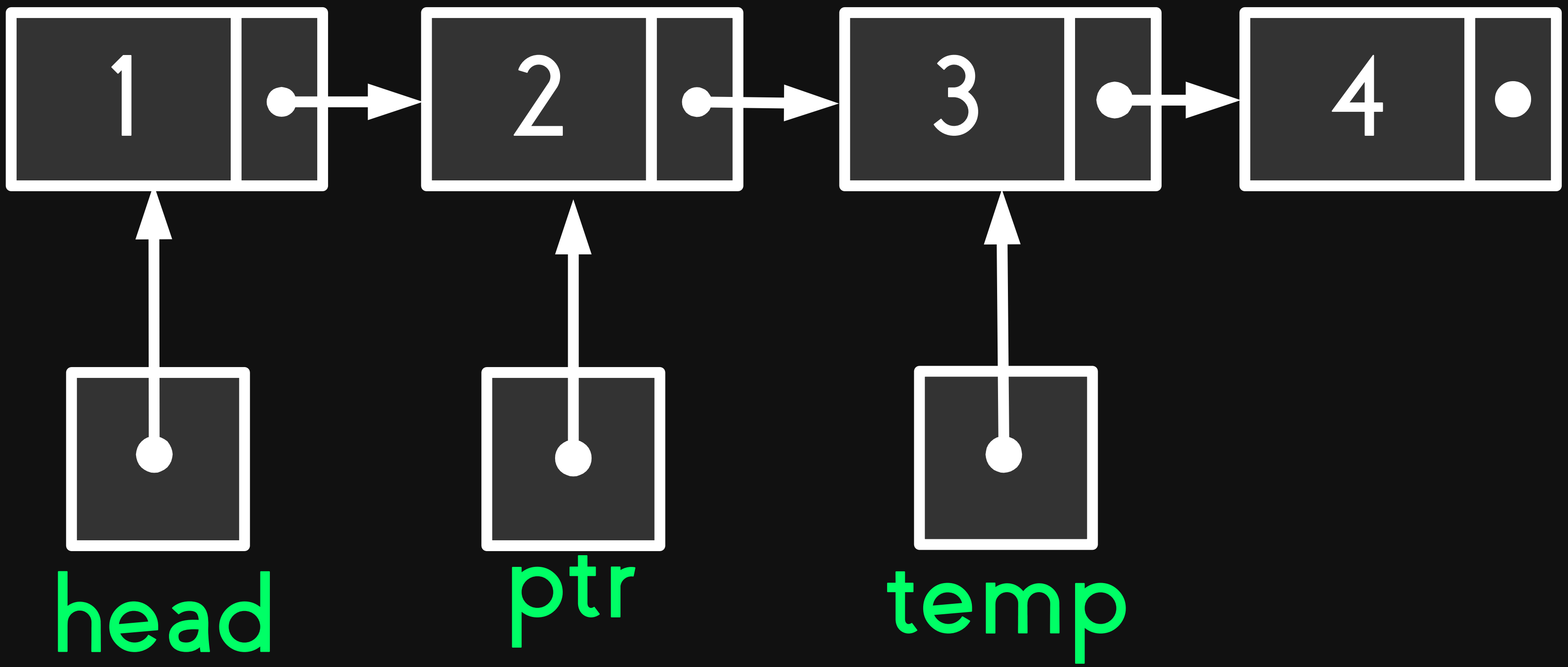
point the `next` pointer of the node being pointed by `ptr` to the new node

Insert at Middle



point the **next** pointer of the node being pointed by **ptr** to the new node

Insert at Middle



Rearrangement of the nodes.

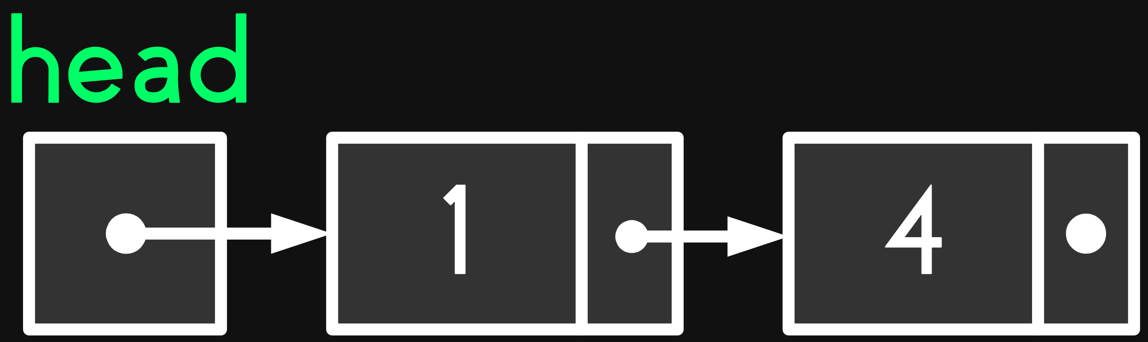
Insert at Tail

can be considered
as a special case of
insert at middle

Insert at Tail

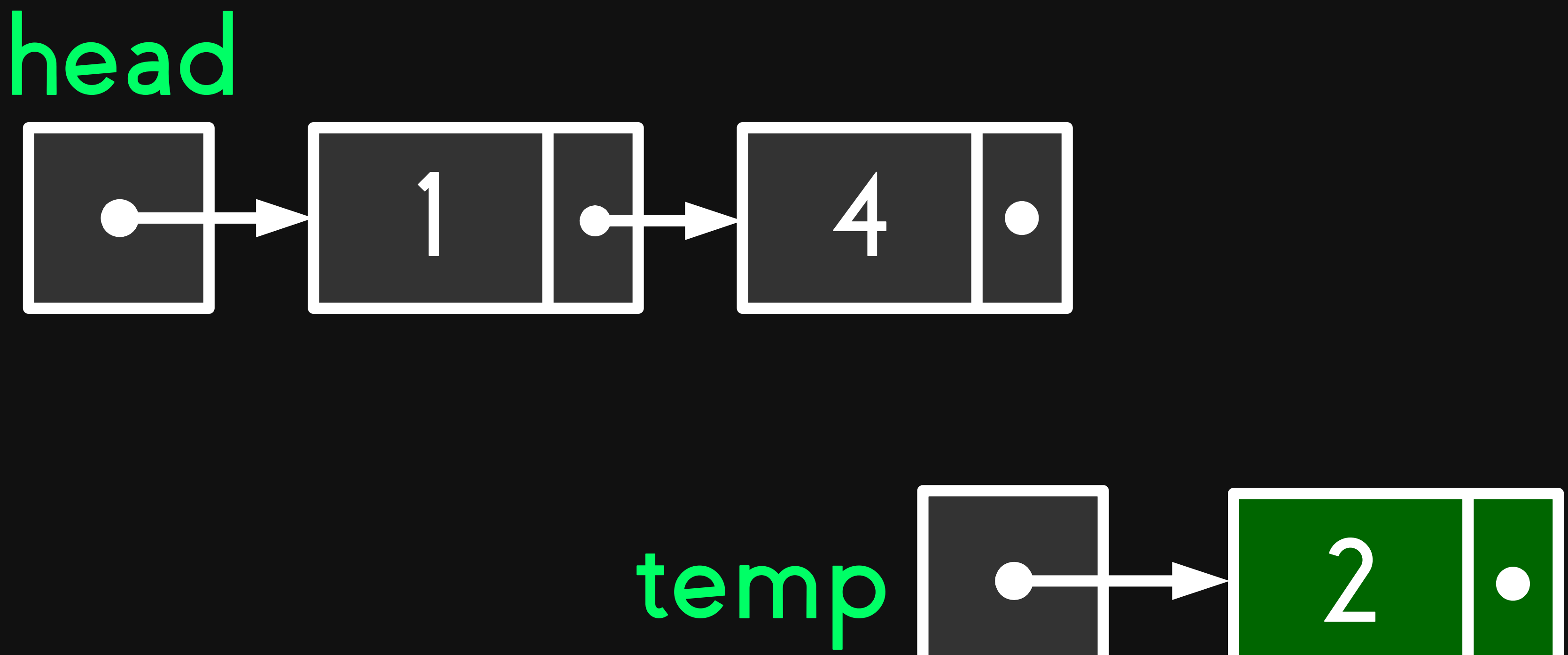
insert a node
after the last node
of the list

Insert at Tail



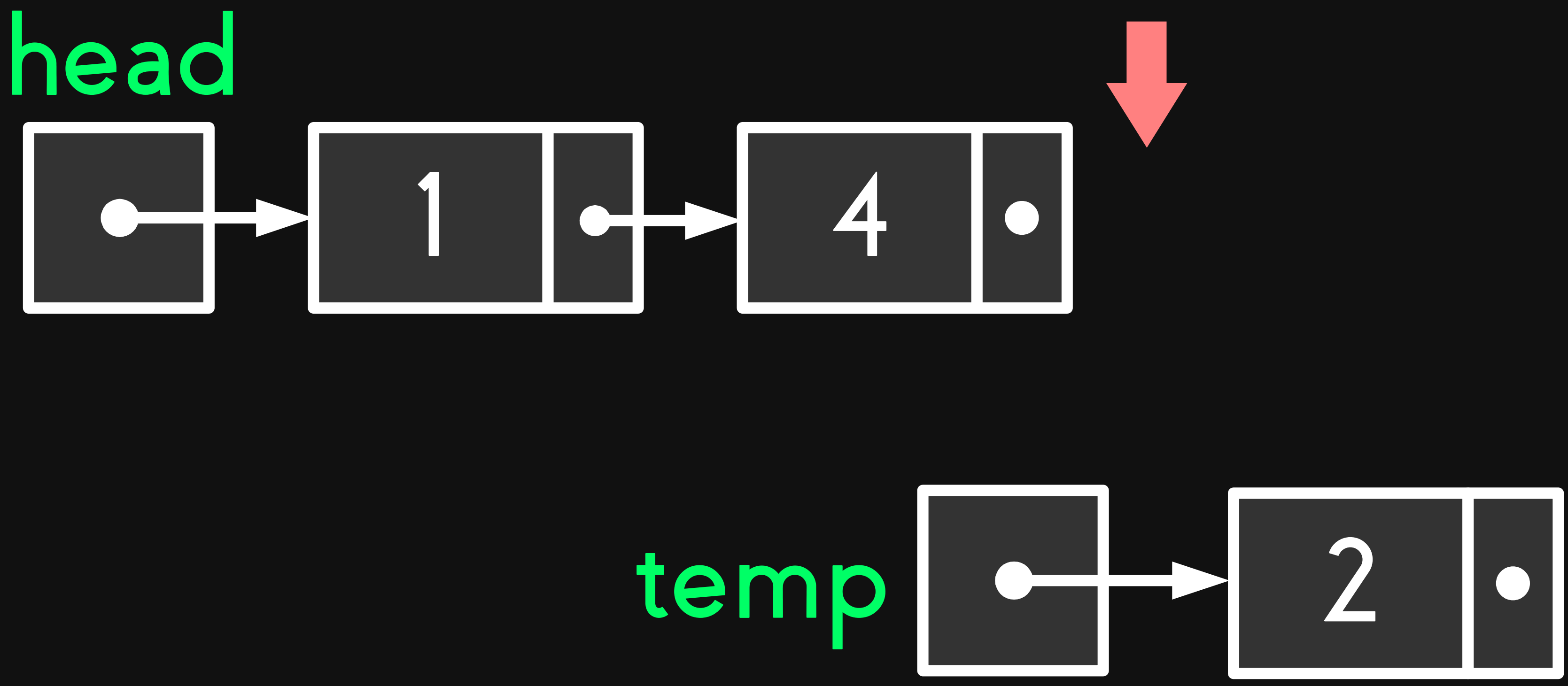
insert 2 at the end
of the linked list.

Insert at Tail



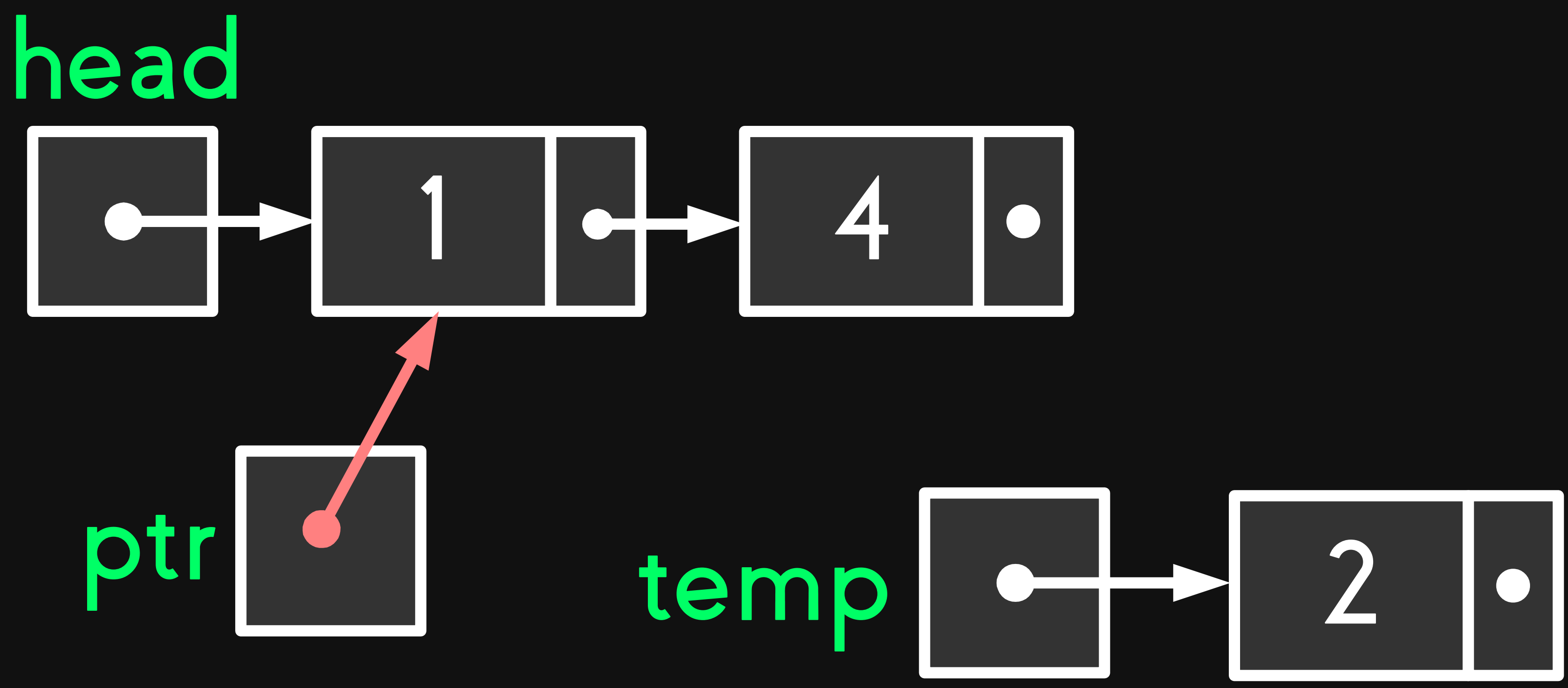
create a new node for 2.
give it to a new pointer (temp).

Insert at Tail



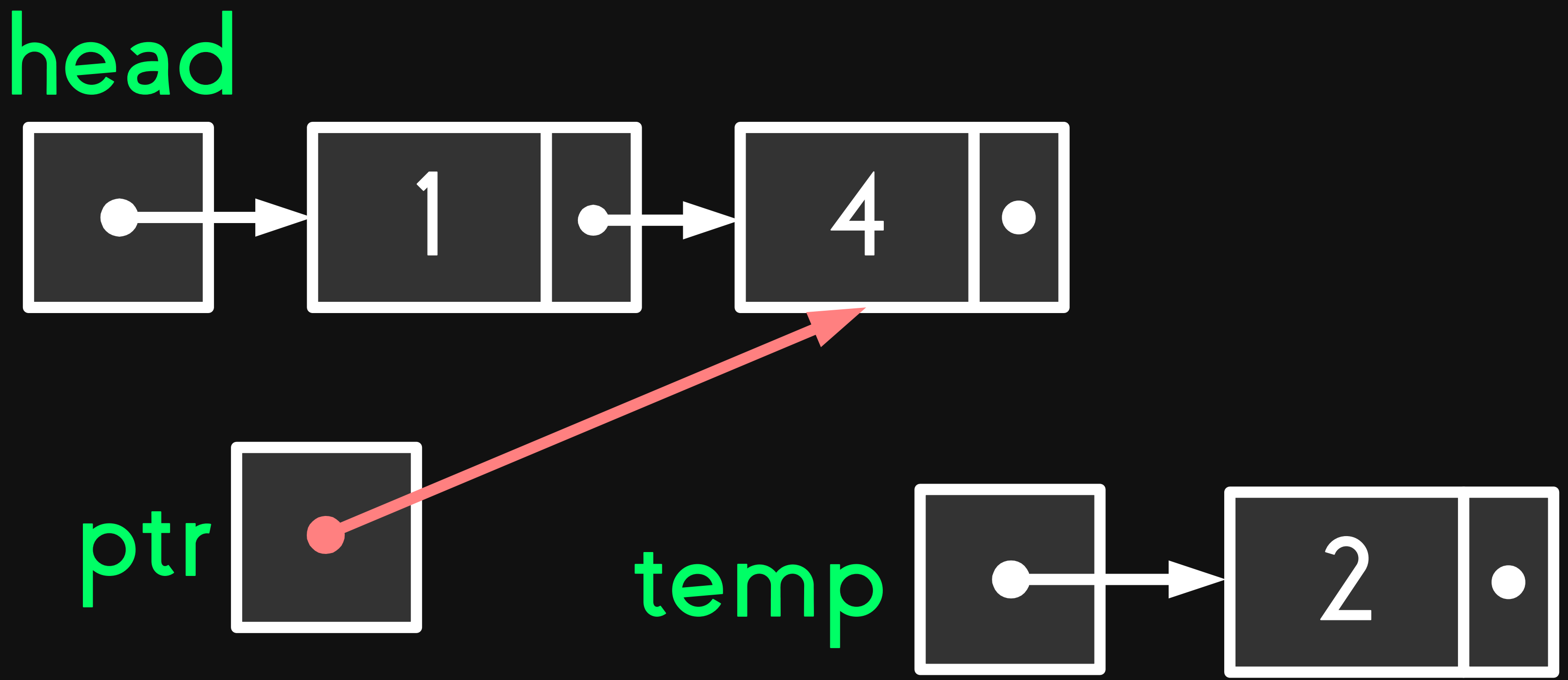
find the tail node.

Insert at Tail



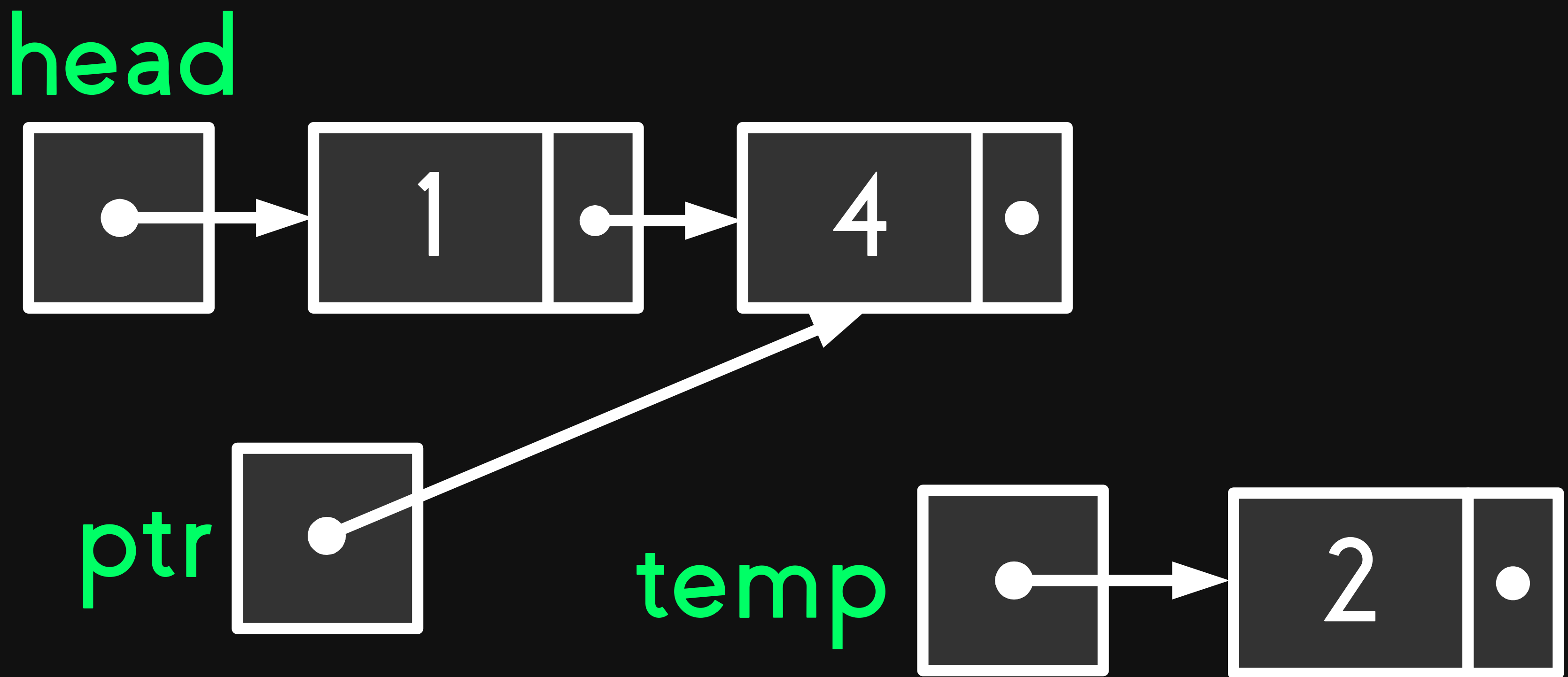
find the tail node.

Insert at Tail



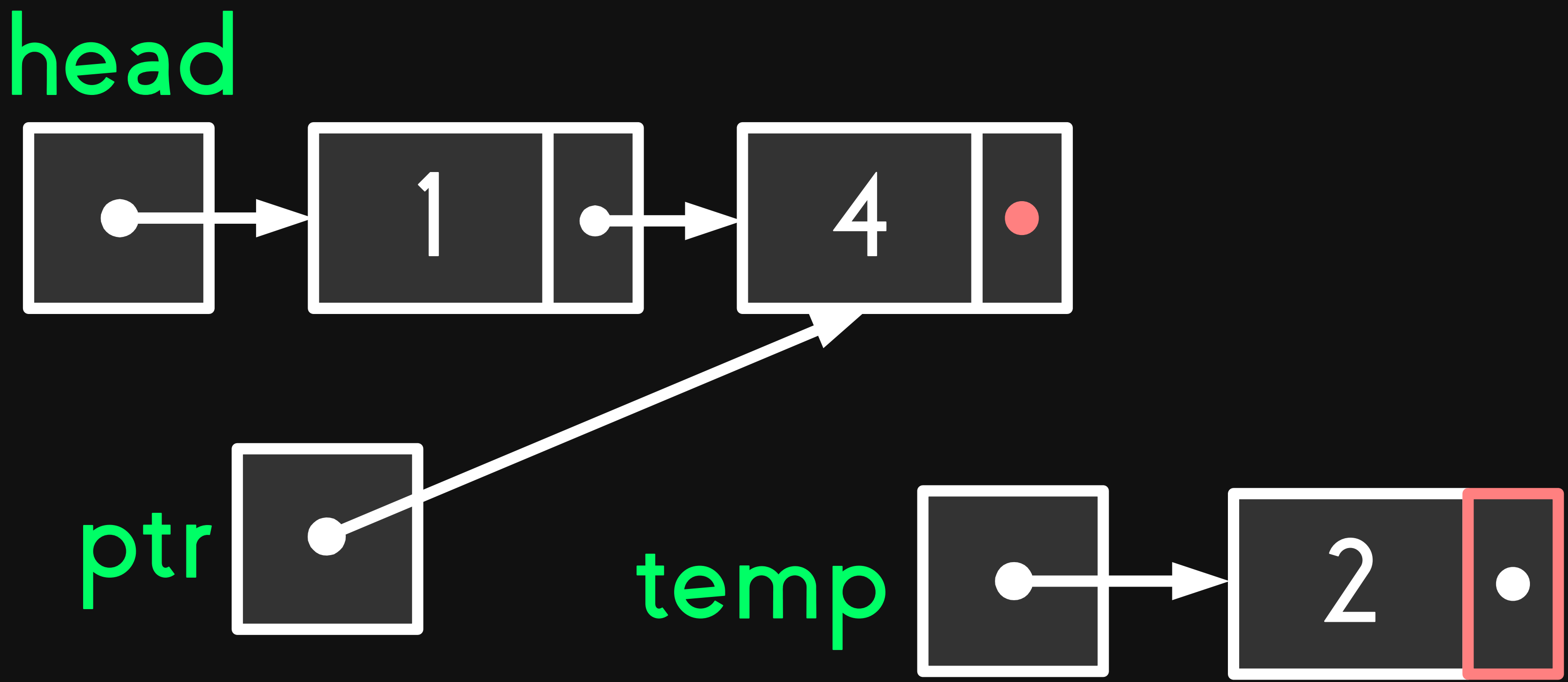
find the tail node.

Insert at Tail



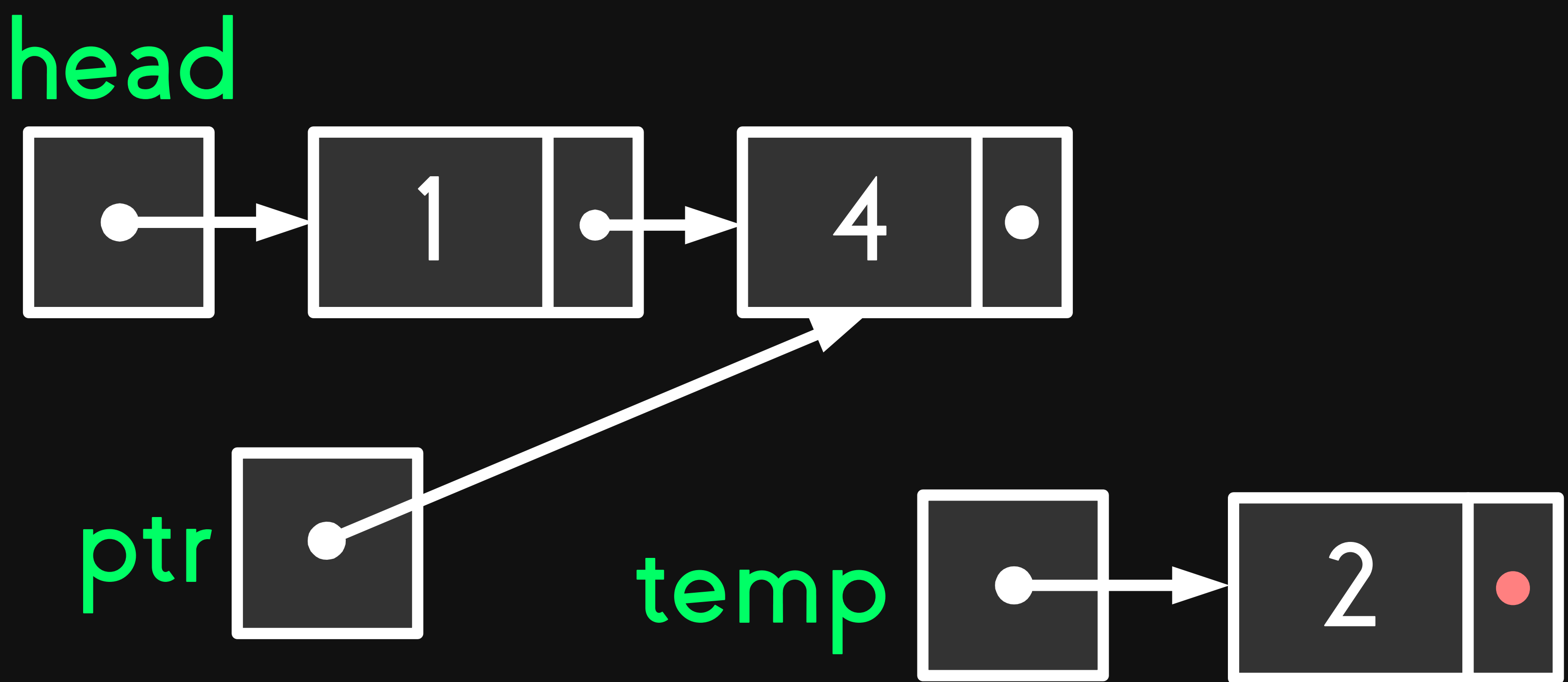
make the next pointer of the new node point to the node being pointed by the next of the node being pointed by ptr.

Insert at Tail



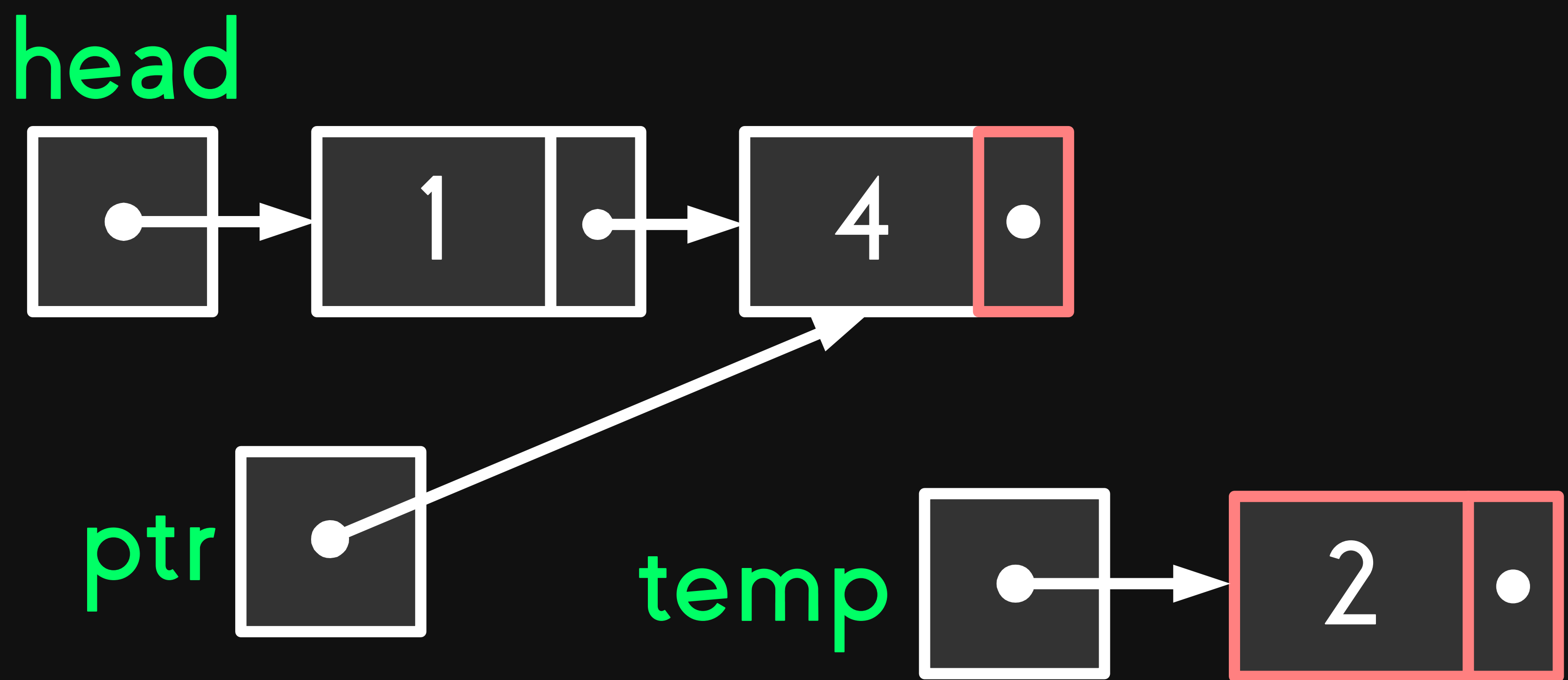
make the next pointer of the new node point to the node being pointed by the next of the node being pointed by ptr.

Insert at Tail



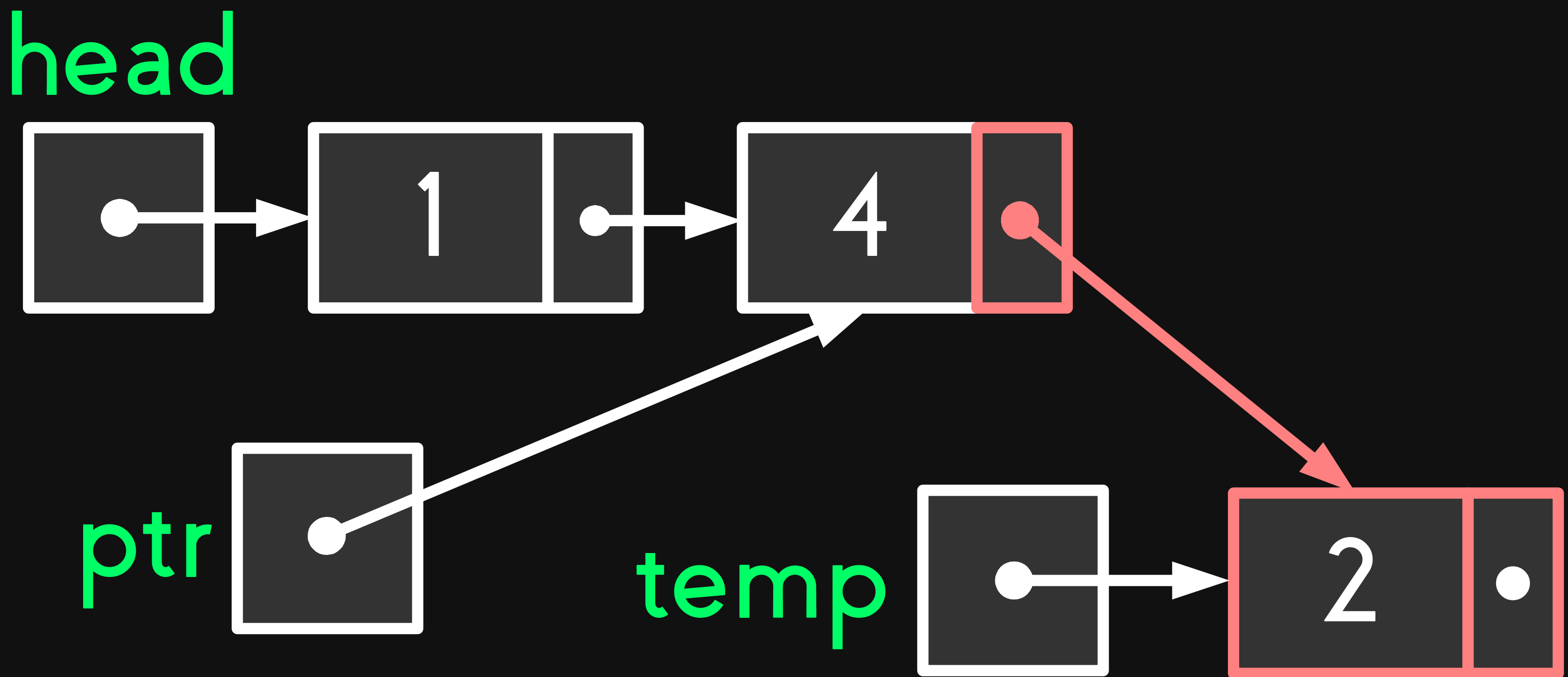
make the next pointer of the new node point to the node being pointed by the next of the node being pointed by ptr.

Insert at Tail



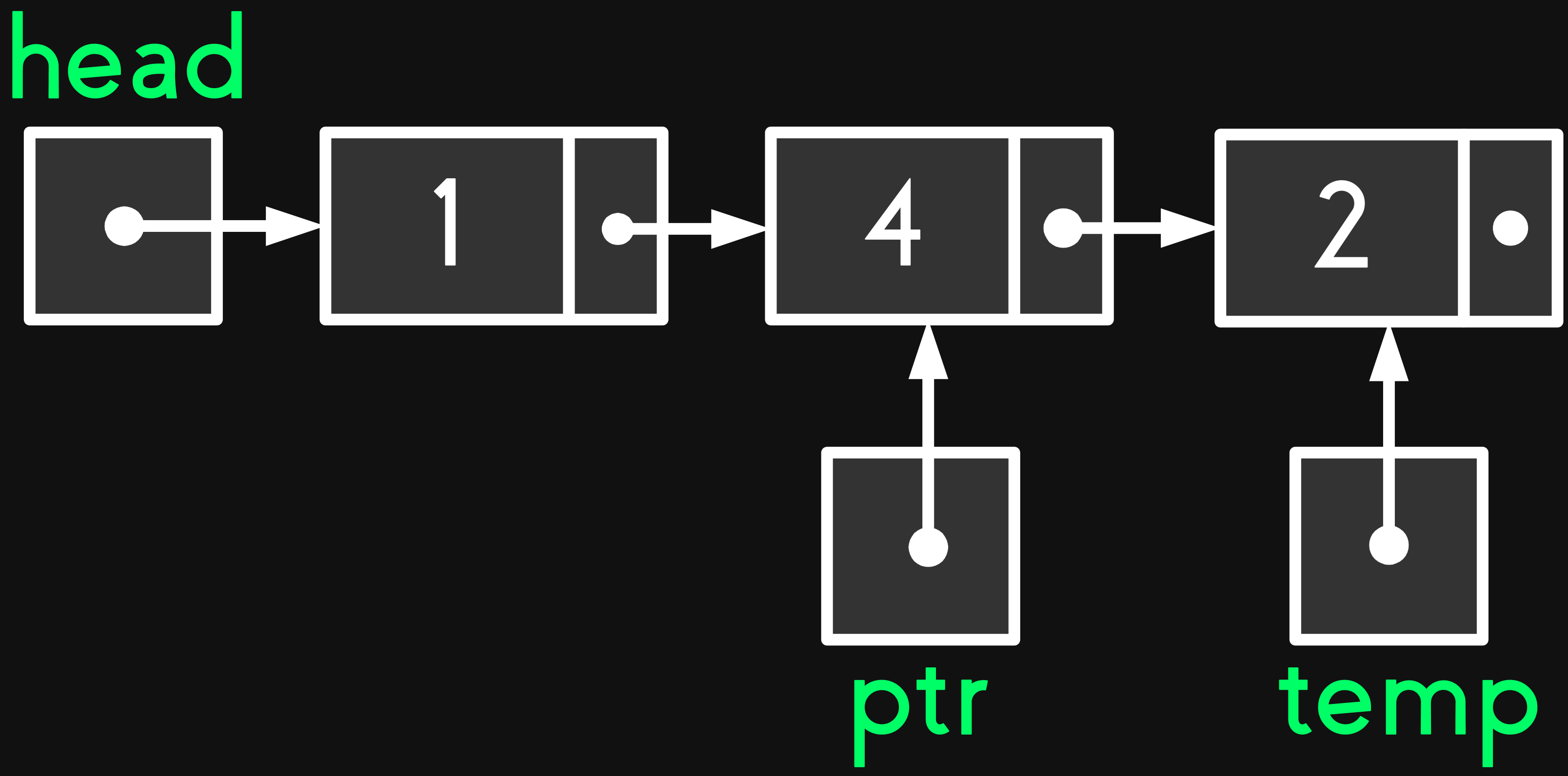
point the `next` of the node being pointed by `ptr` to the new node

Insert at Tail



point the `next` of the node being pointed by `ptr` to the new node

Insert at Tail



Rearrangement of the list.

Delete

delete nodes from a linked list

Delete

delete nodes from a linked list

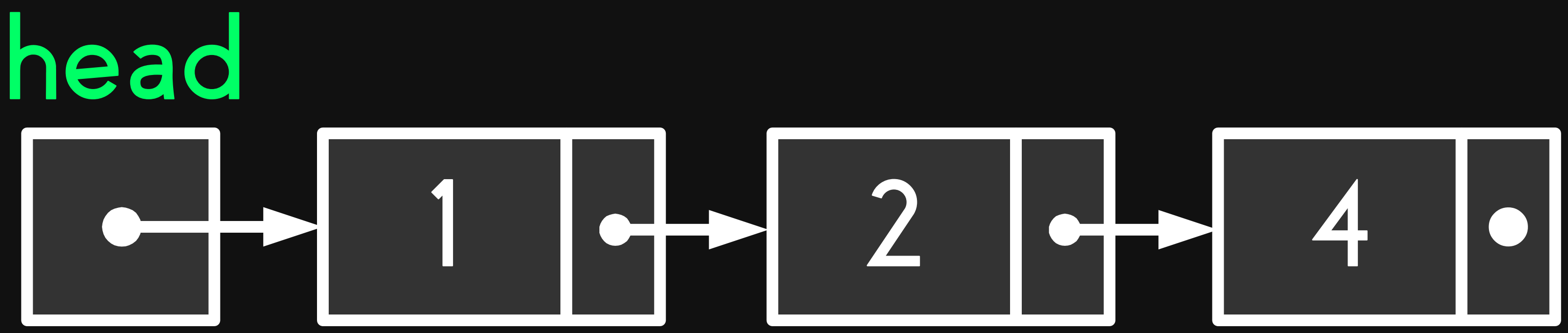
has three(3) cases:

- delete at head
- delete at middle
- delete at tail

Delete at Head

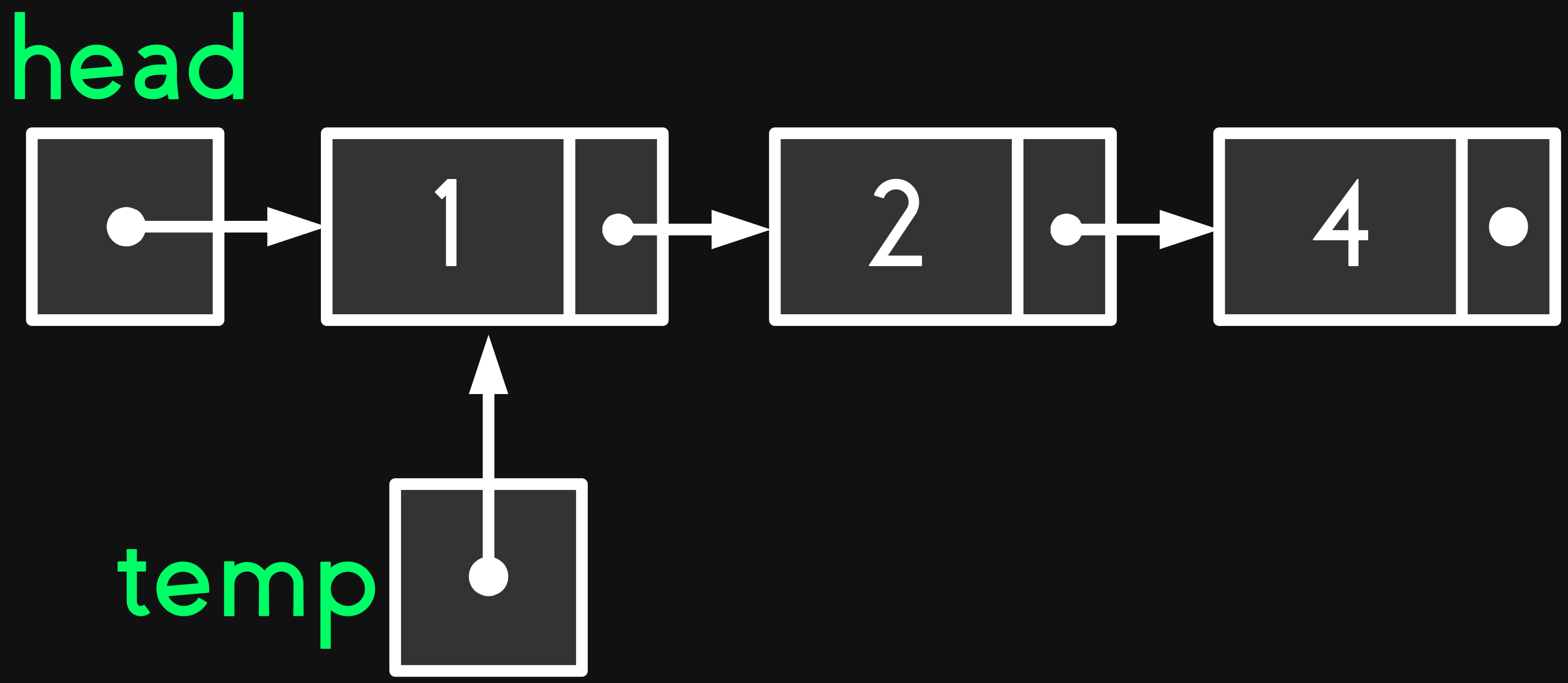
delete the
first element
of the list

Delete at Head



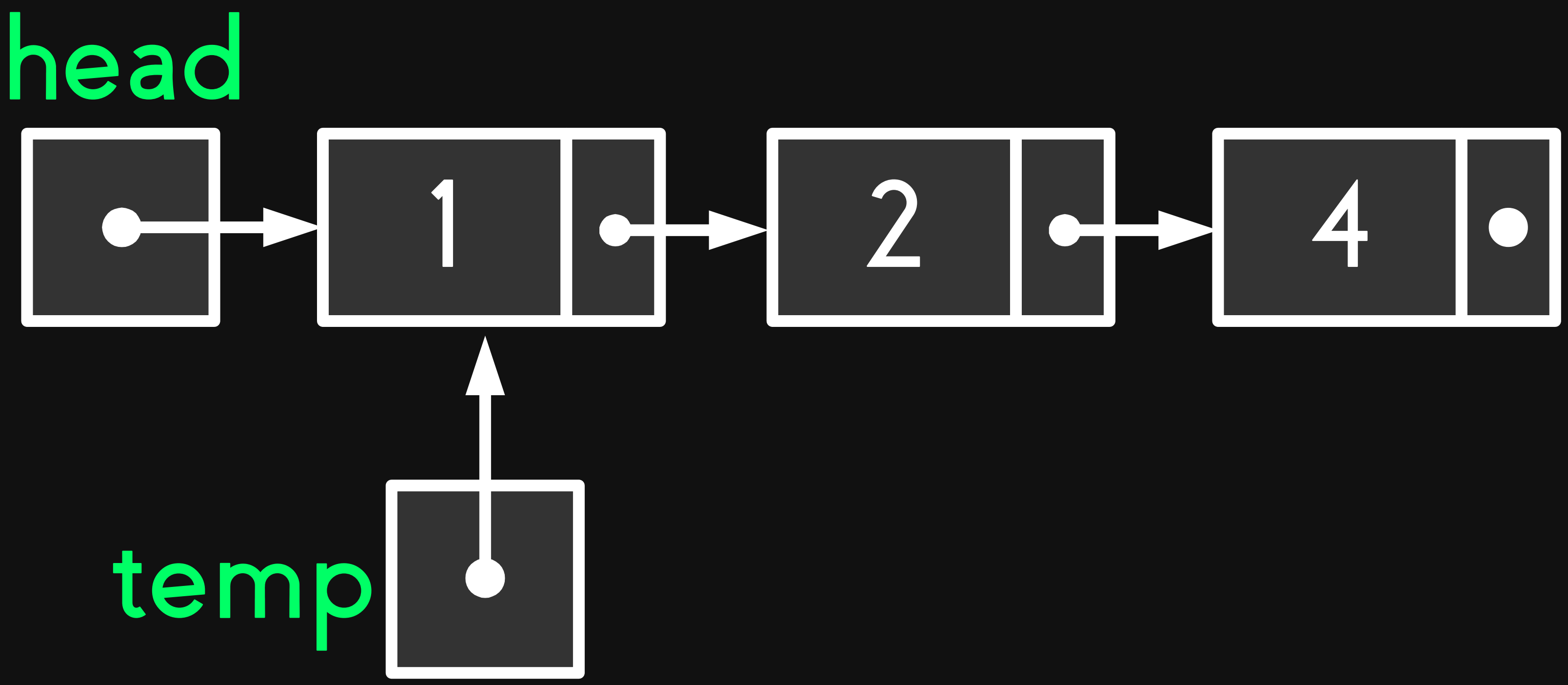
delete the first element (1)

Delete at Head



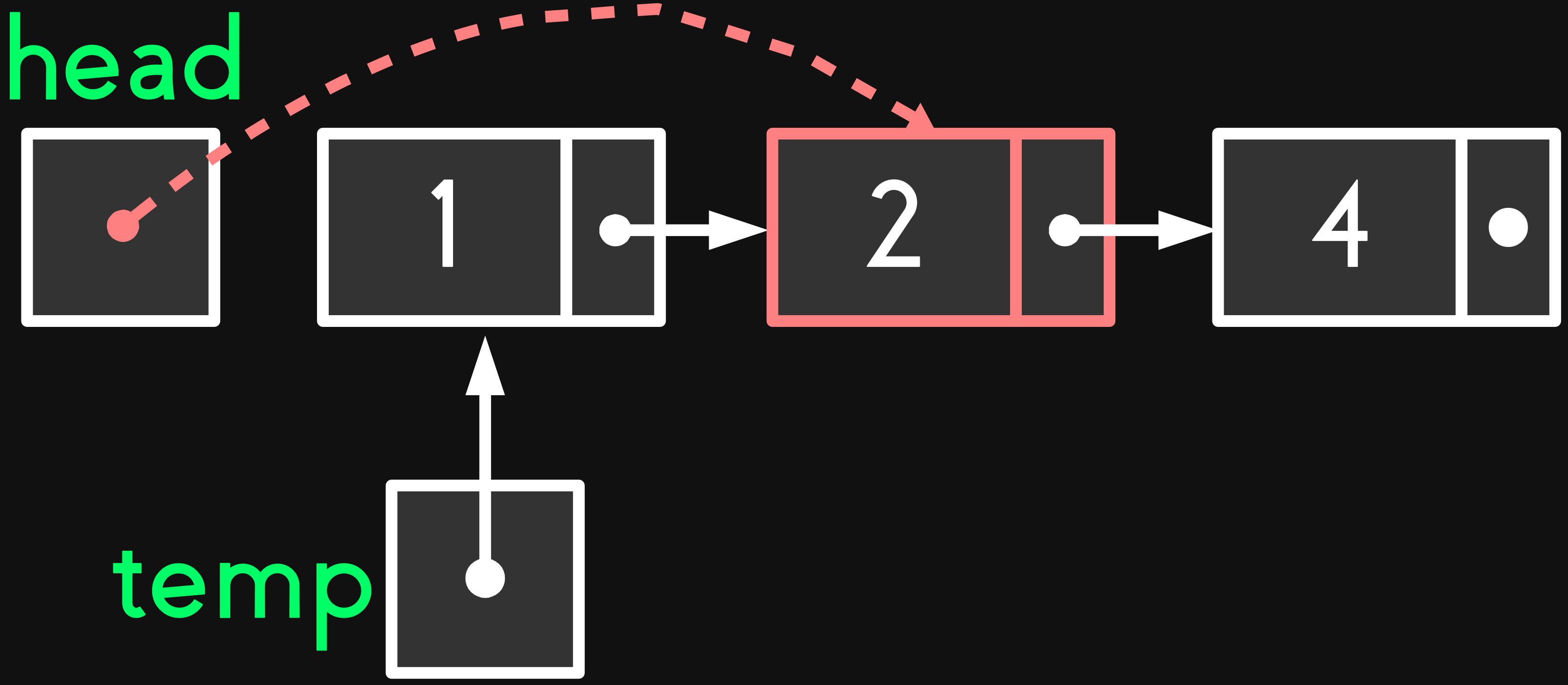
make a pointer (`temp`) point to
the node to be deleted

Delete at Head



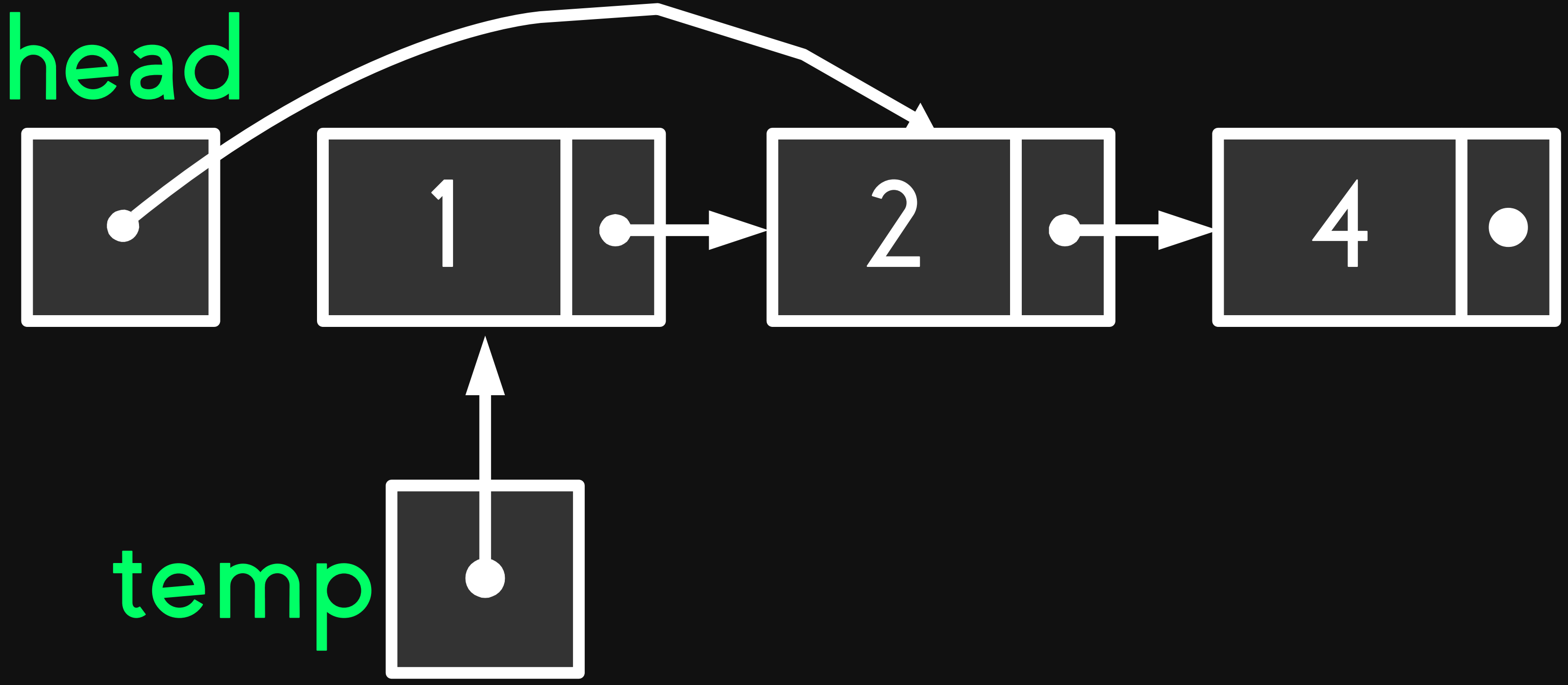
point head to the node after the node to be deleted

Delete at Head

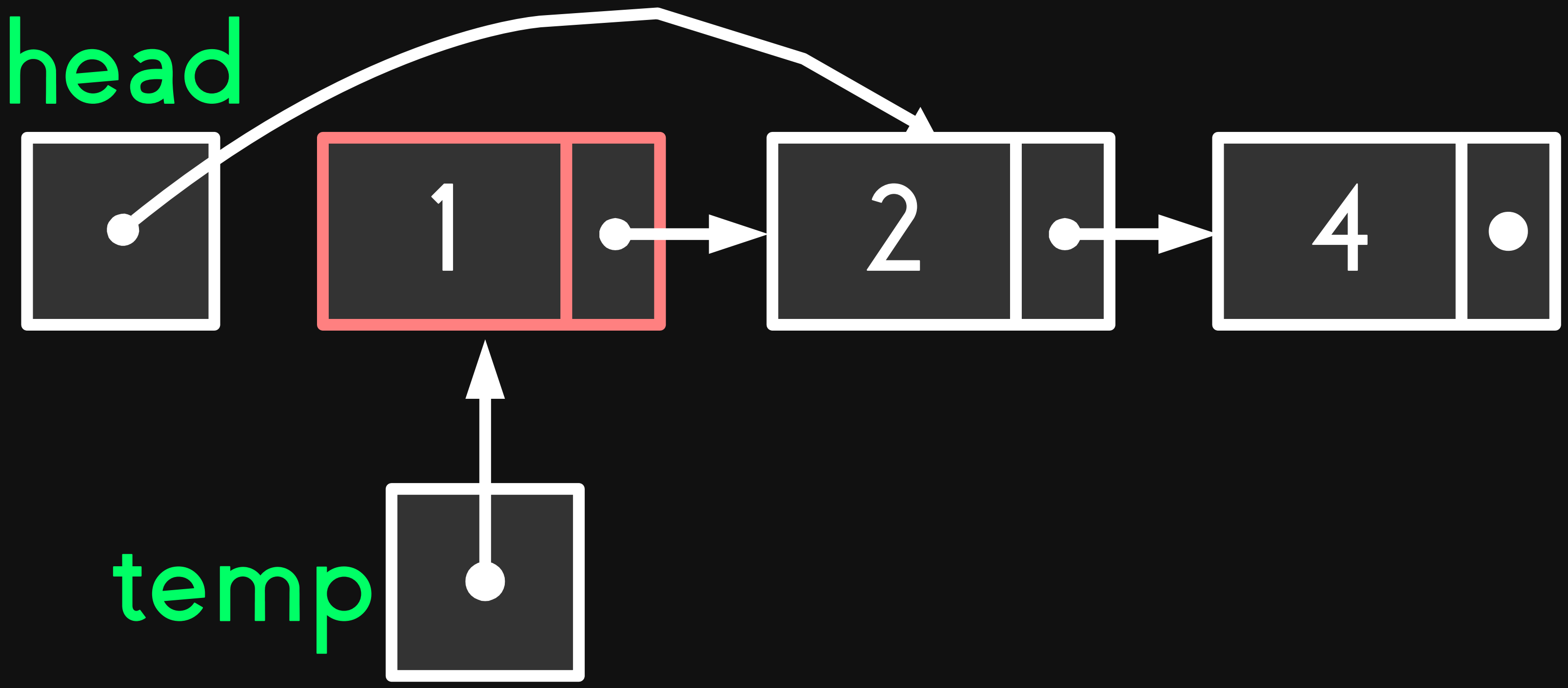


point head to the node after the node to be deleted

Delete at Head

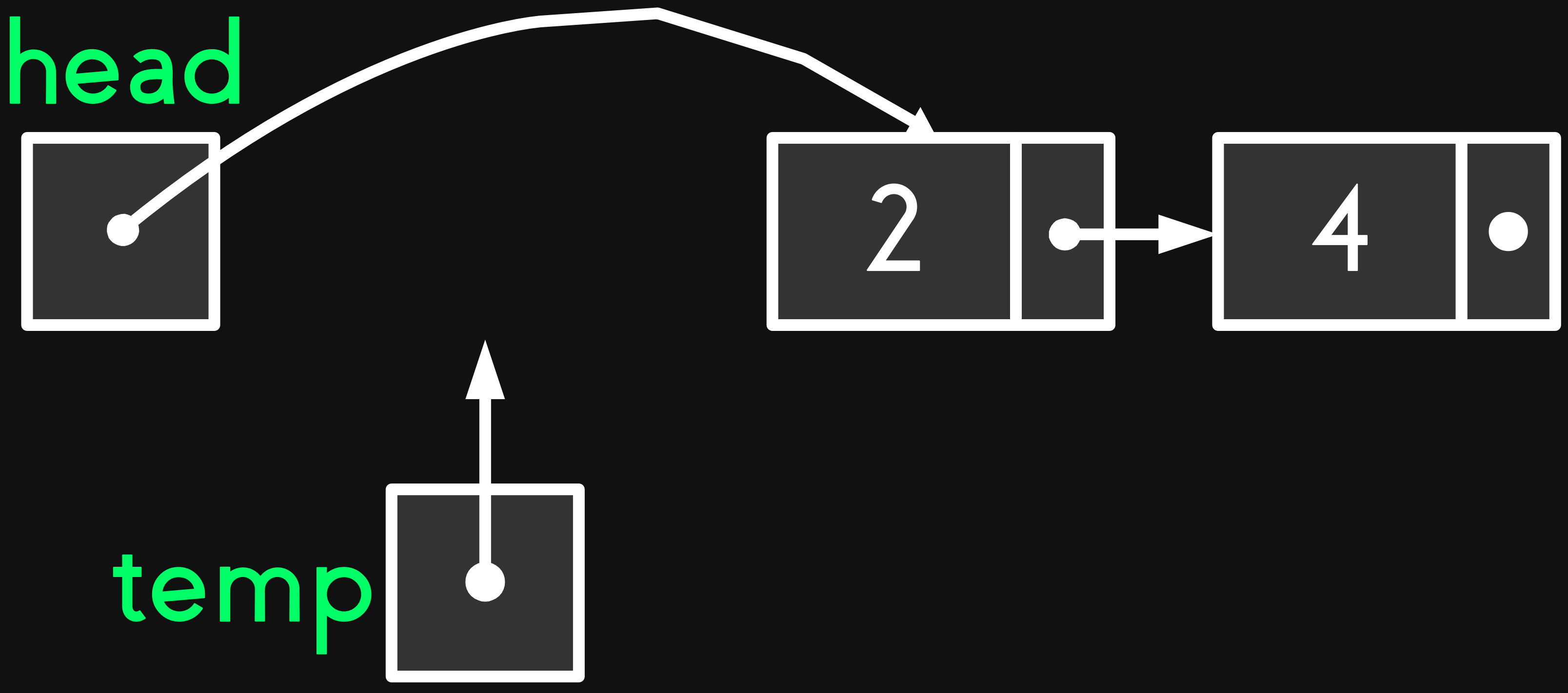


Delete at Head



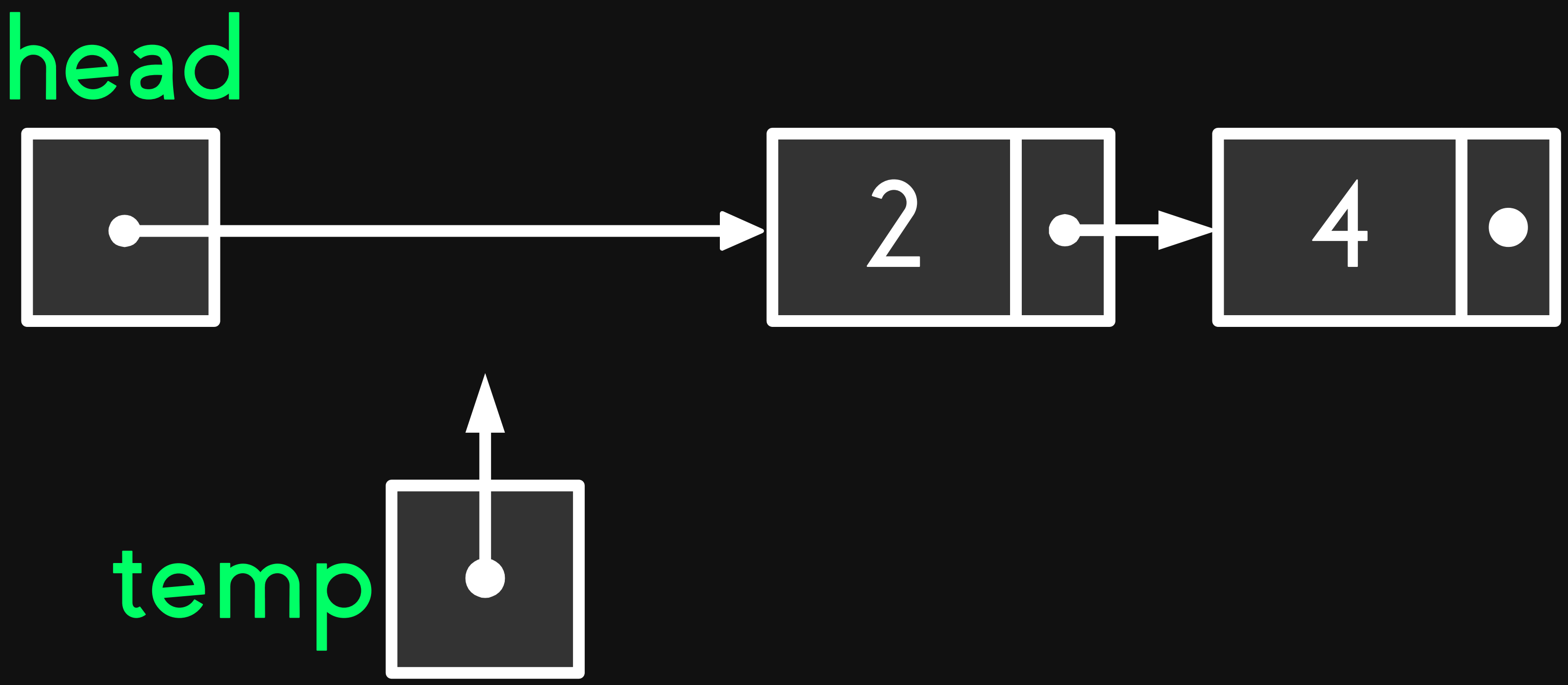
free the node being
pointed by temp

Delete at Head



free the node being
pointed by temp

Delete at Head



`temp` is now a dangling pointer.

Delete at Middle

delete a node that
is **in between two nodes**
in the linked list

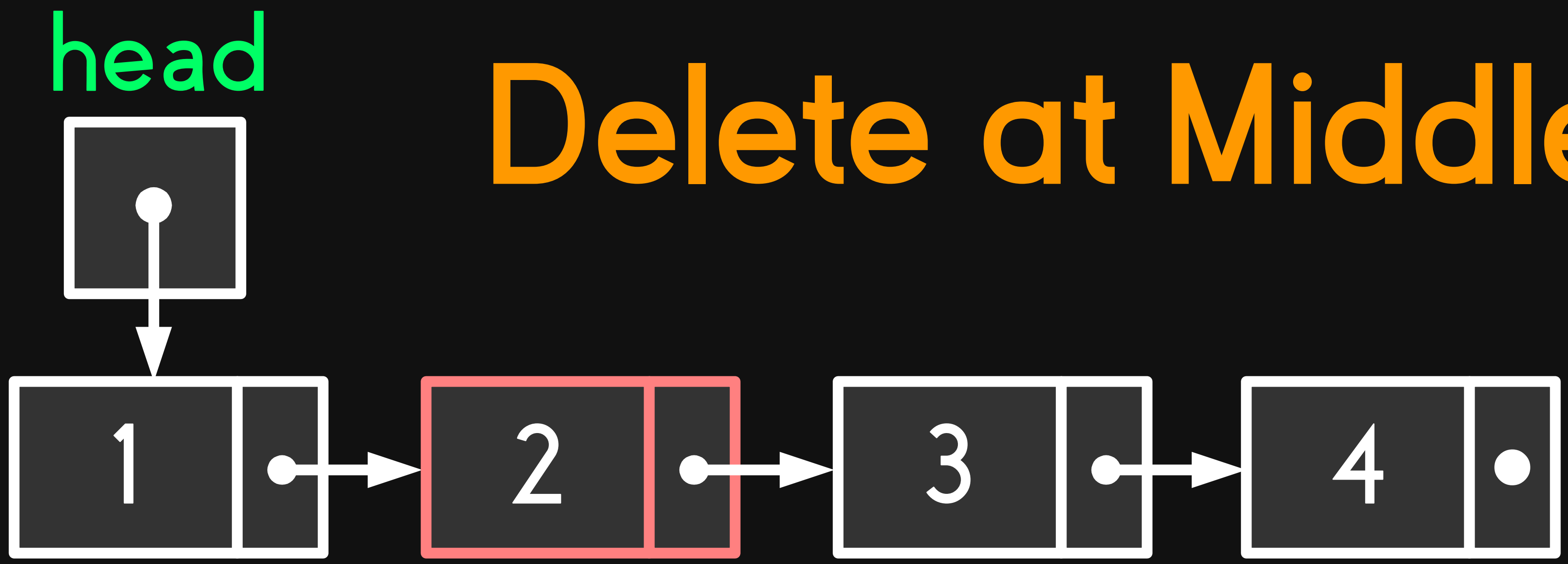
Delete at Middle

delete the node containing 3.

Delete at Middle

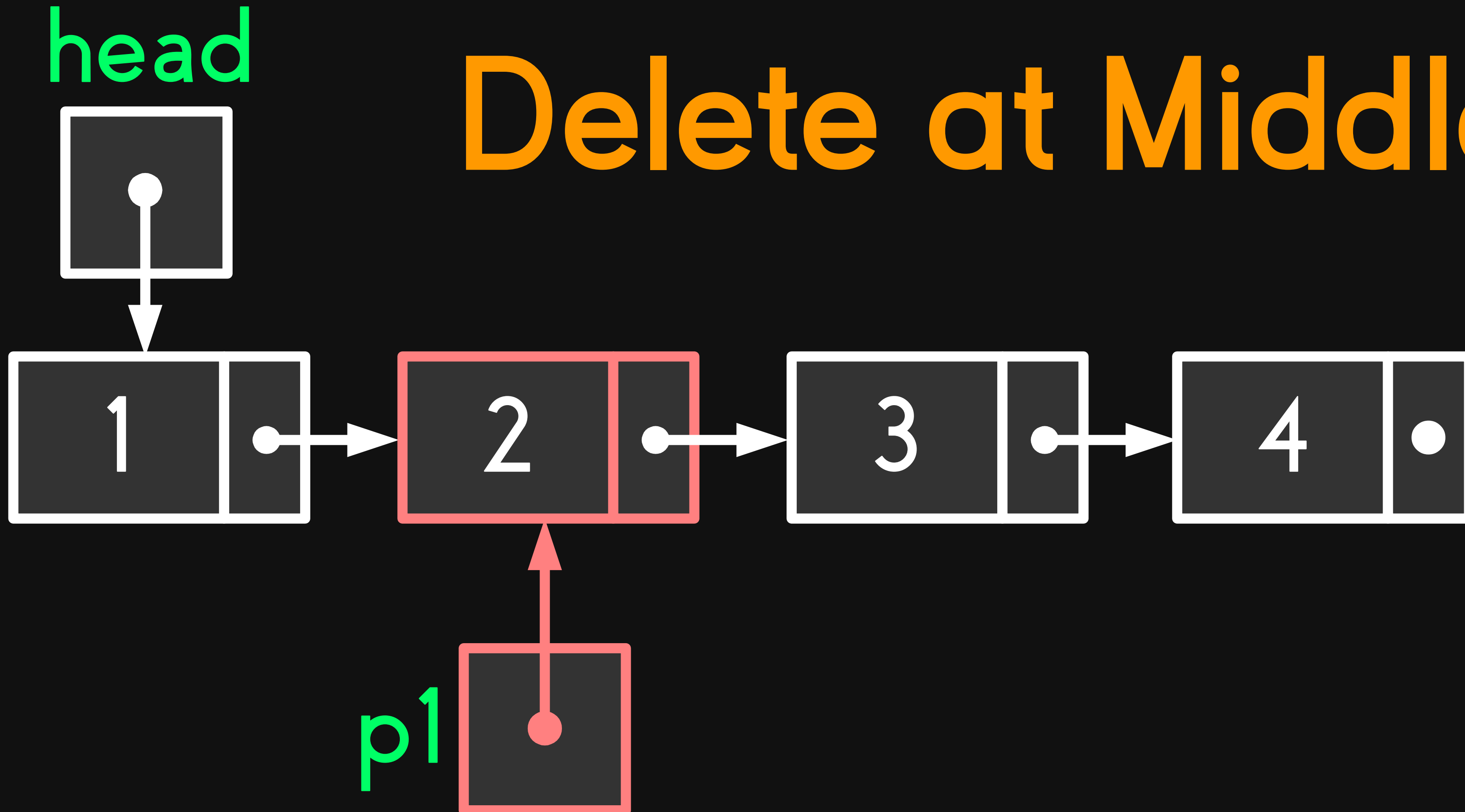
find the node before the node to be deleted.

Delete at Middle



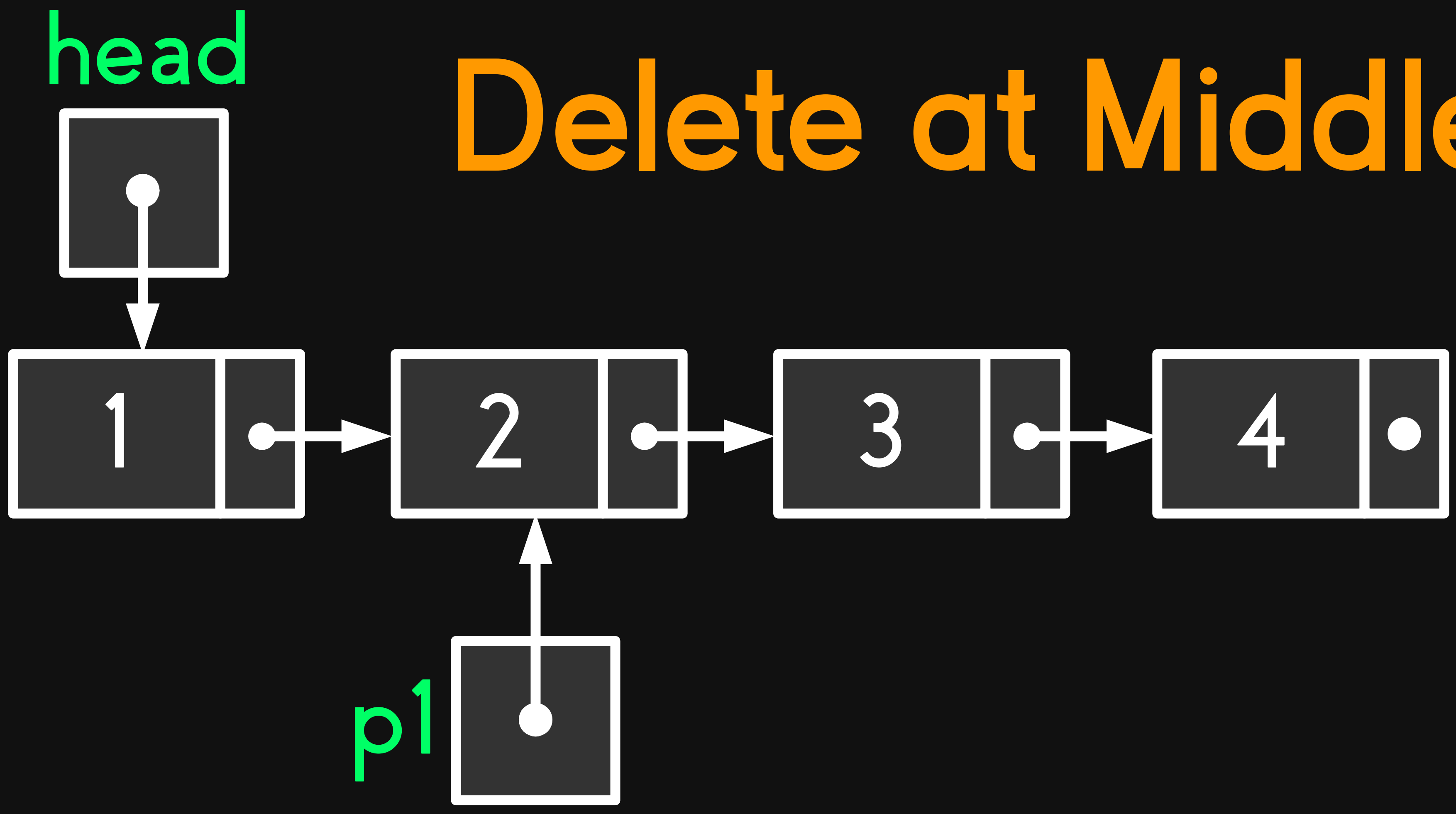
find the node before the node to be deleted.

Delete at Middle



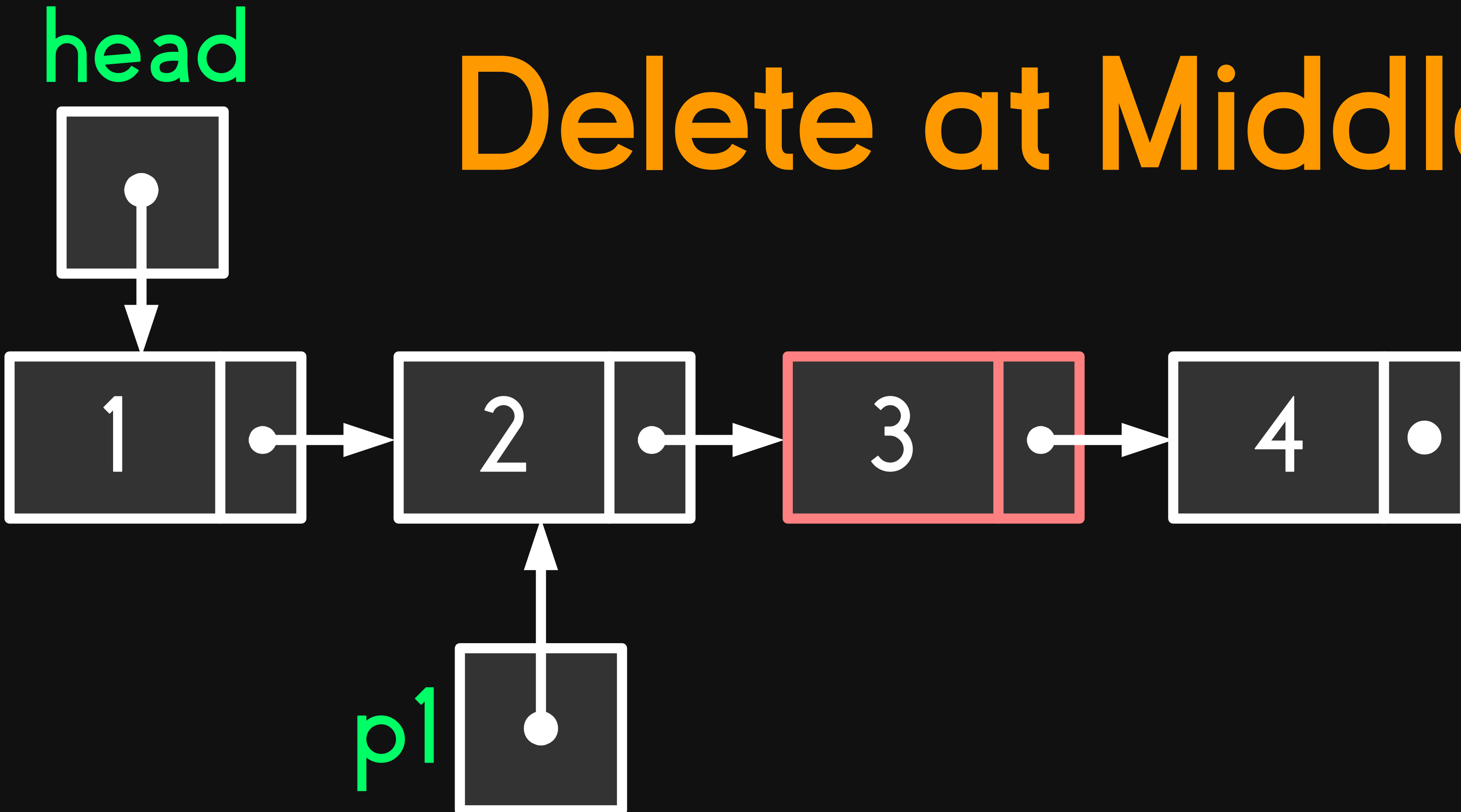
point a pointer (**p1**) to the node
before the node to be deleted.

Delete at Middle



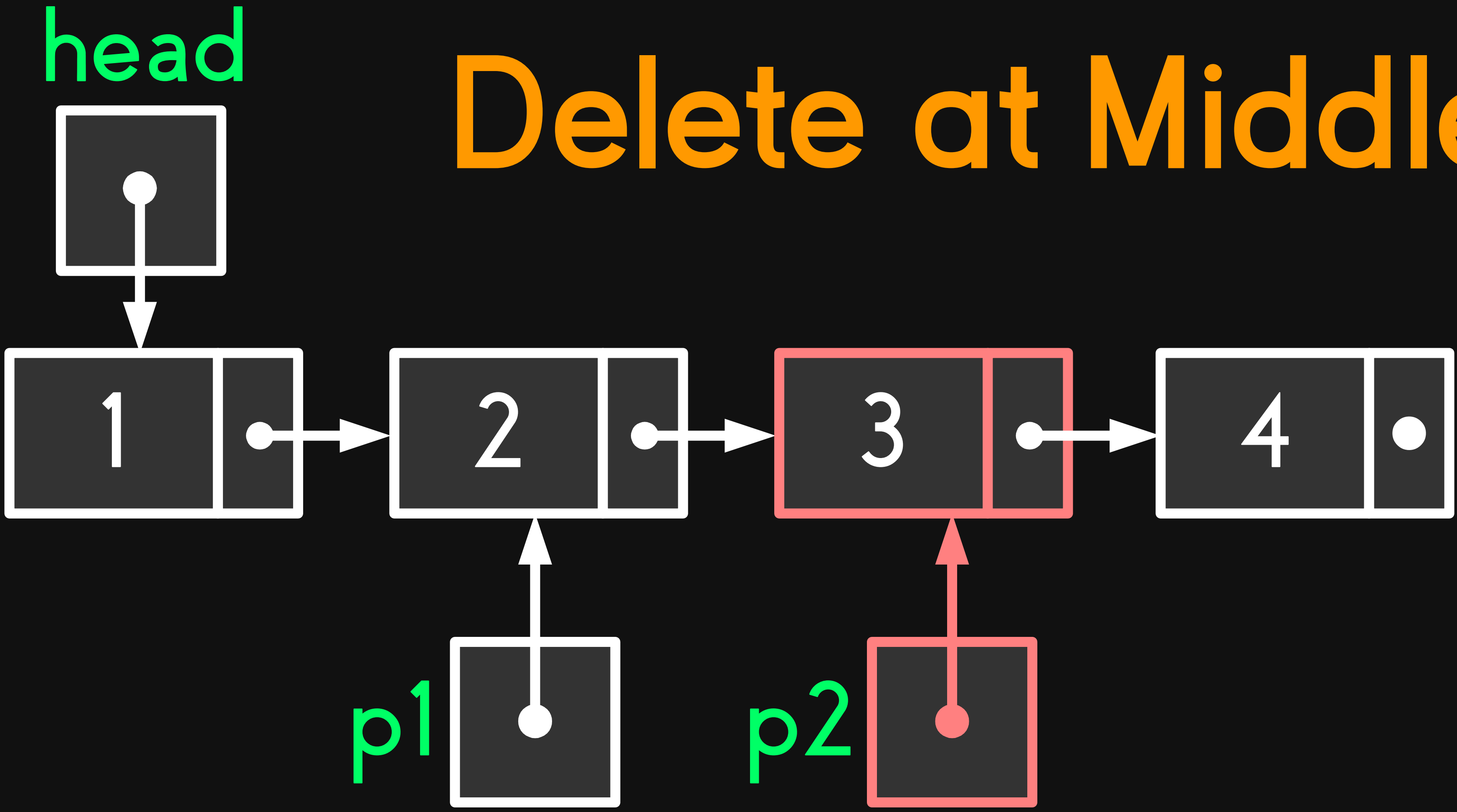
point another pointer (**p2**) to the node to be deleted.

Delete at Middle



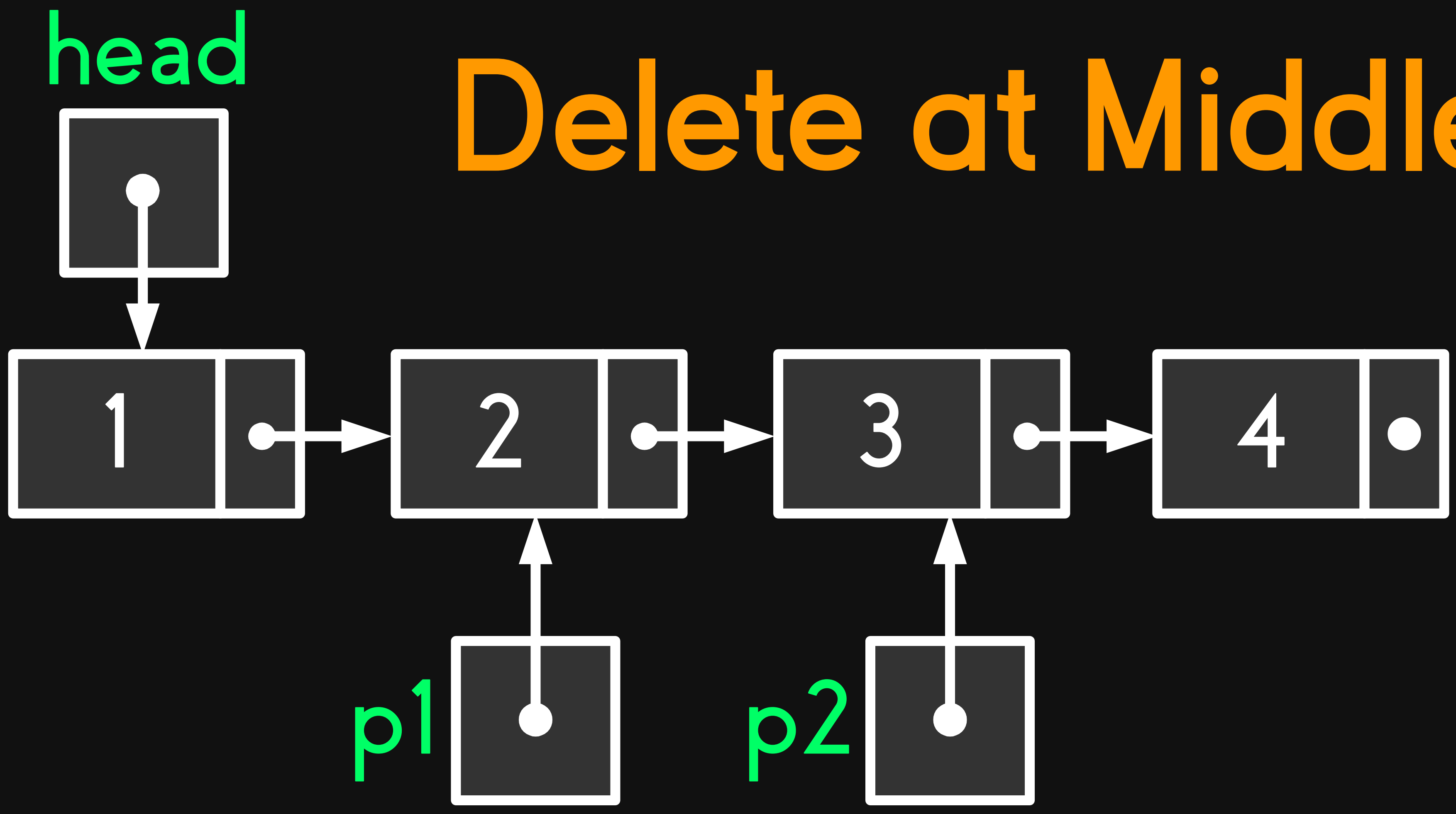
point another pointer (**p2**) to the node to be deleted.

Delete at Middle

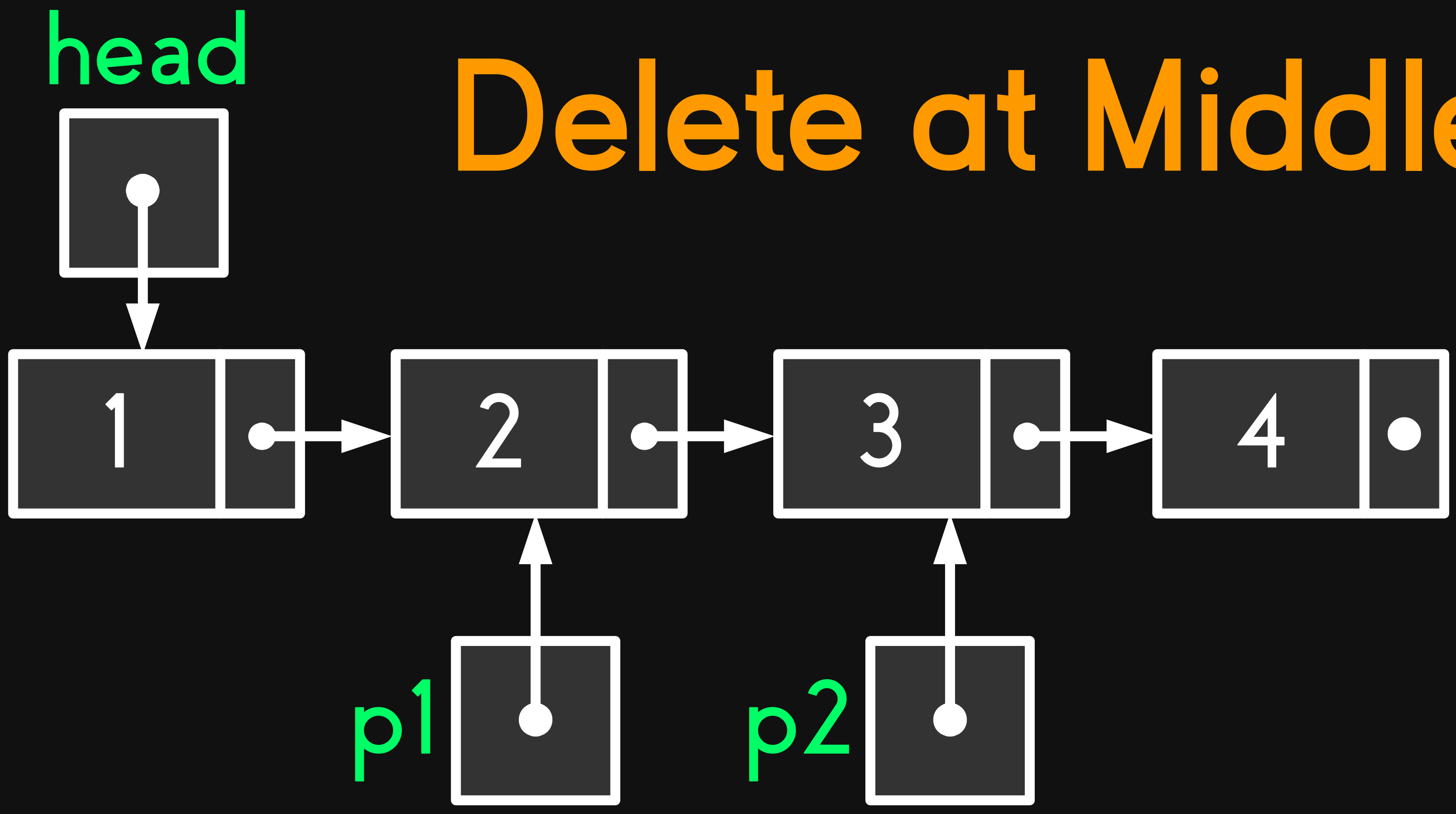


point another pointer (**p2**) to the
node to be deleted.

Delete at Middle

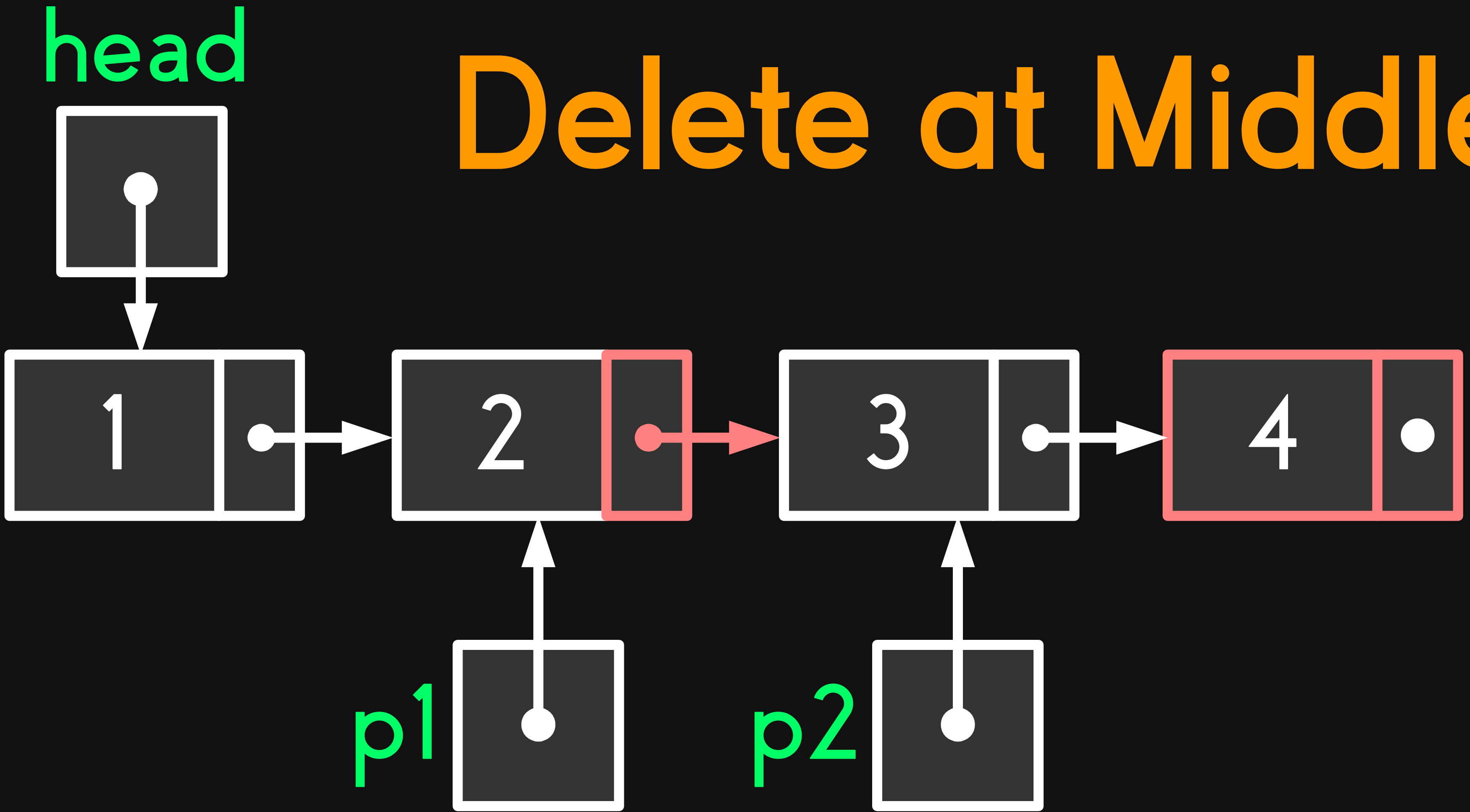


Delete at Middle



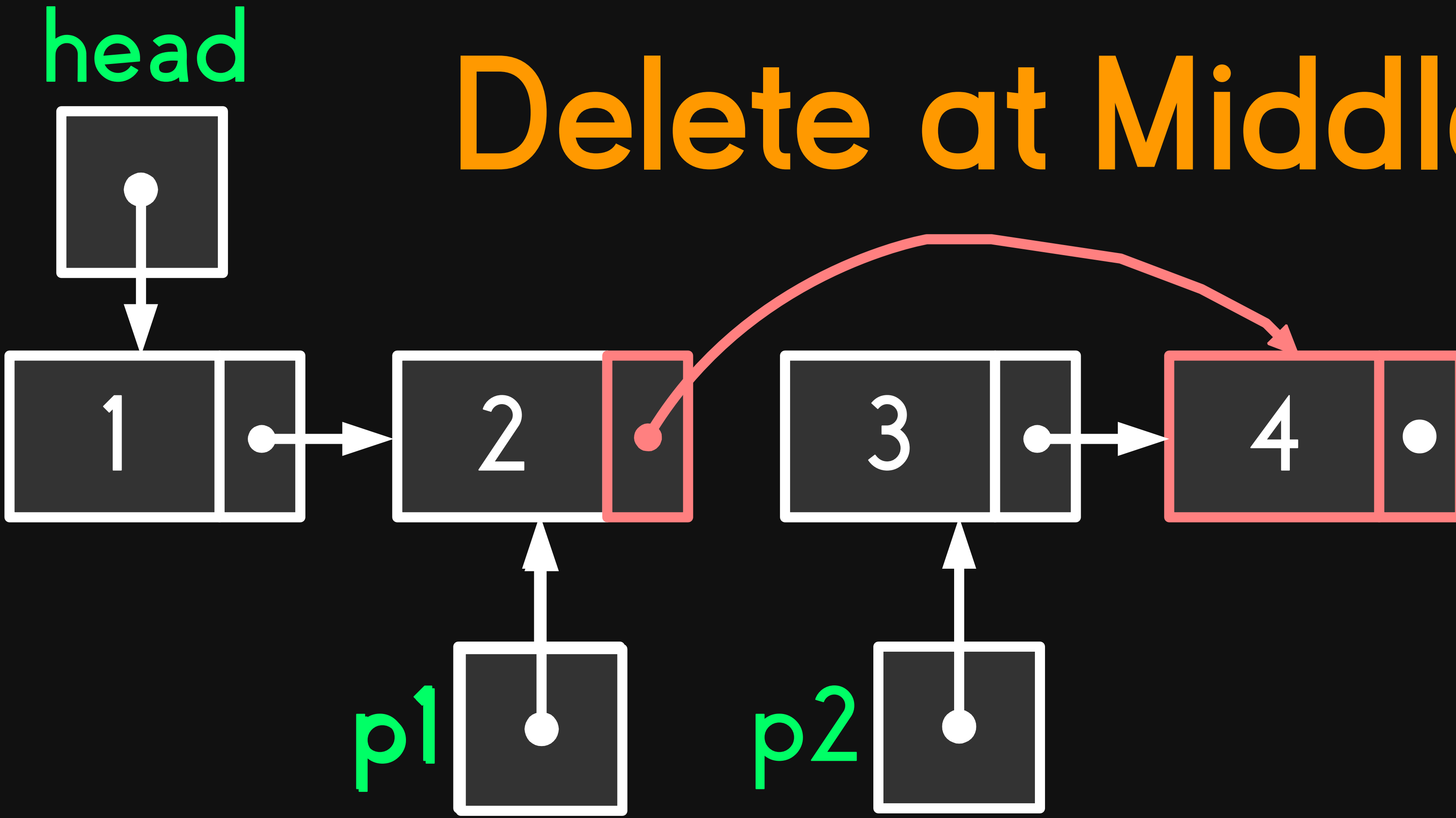
point the next pointer of the node being pointed by **p1** to the node after the node to be deleted

Delete at Middle



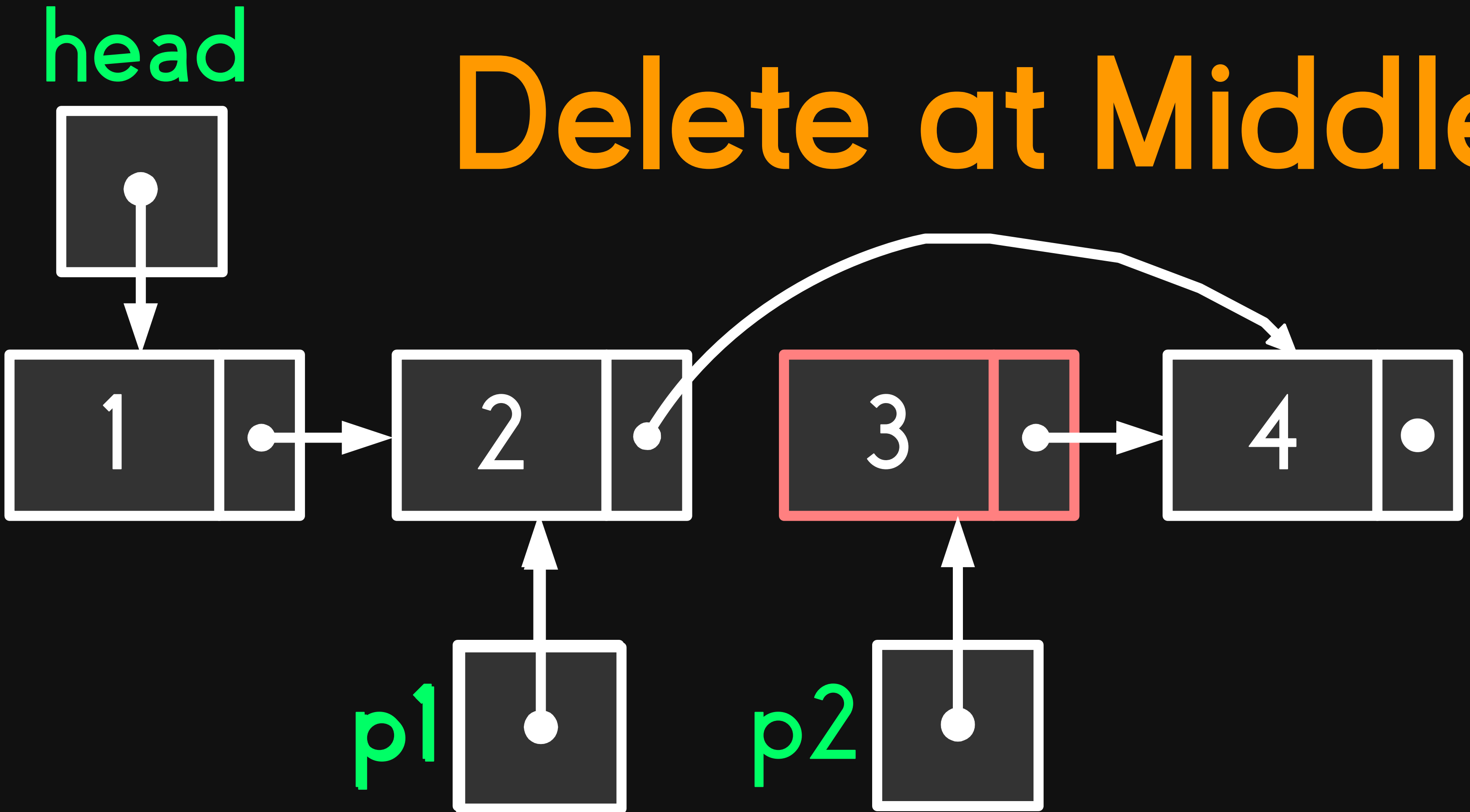
point the next pointer of the node being pointed by **p1** to the node after the node to be deleted

Delete at Middle



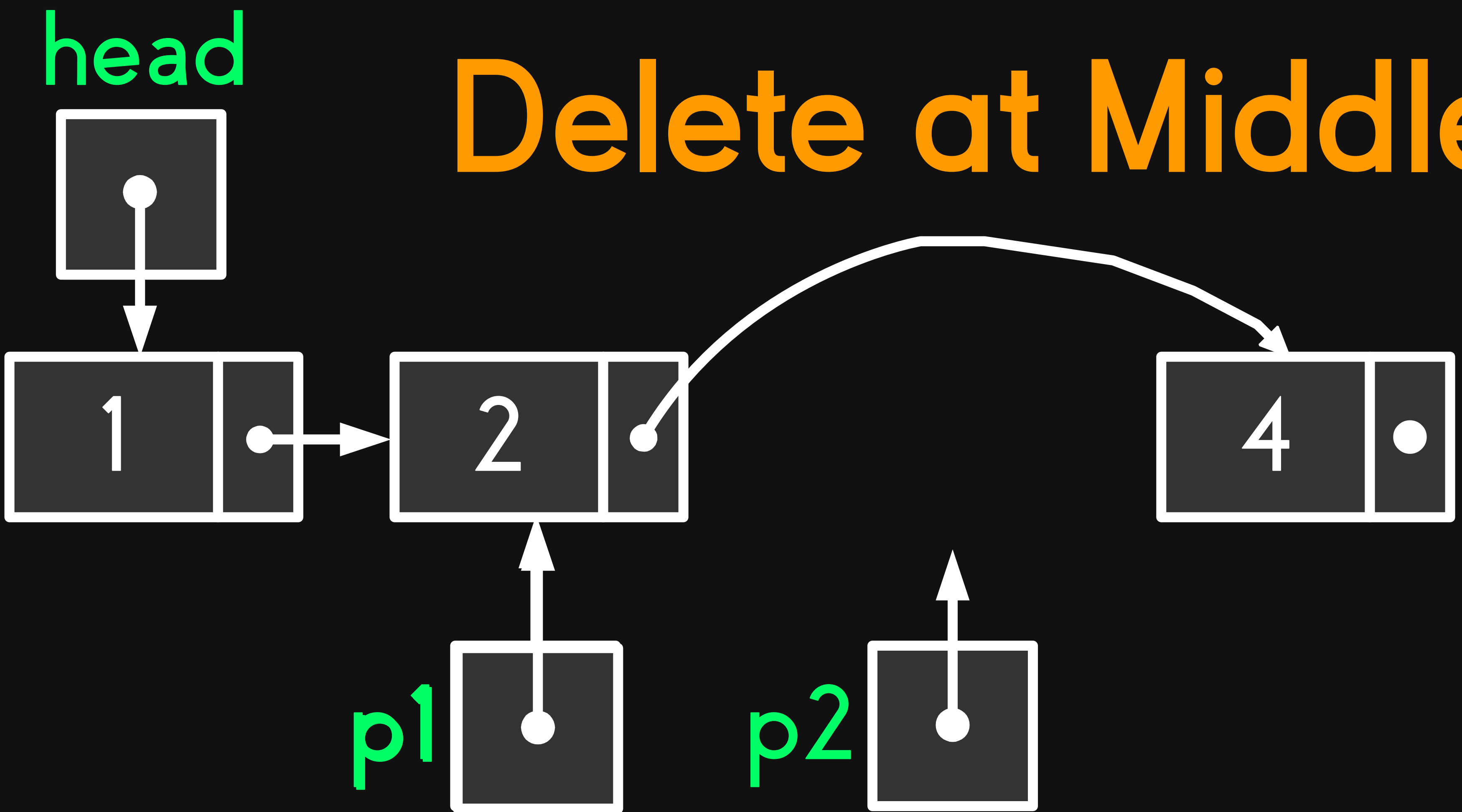
point the next pointer of the node being pointed by **p1** to the node after the node to be deleted

Delete at Middle



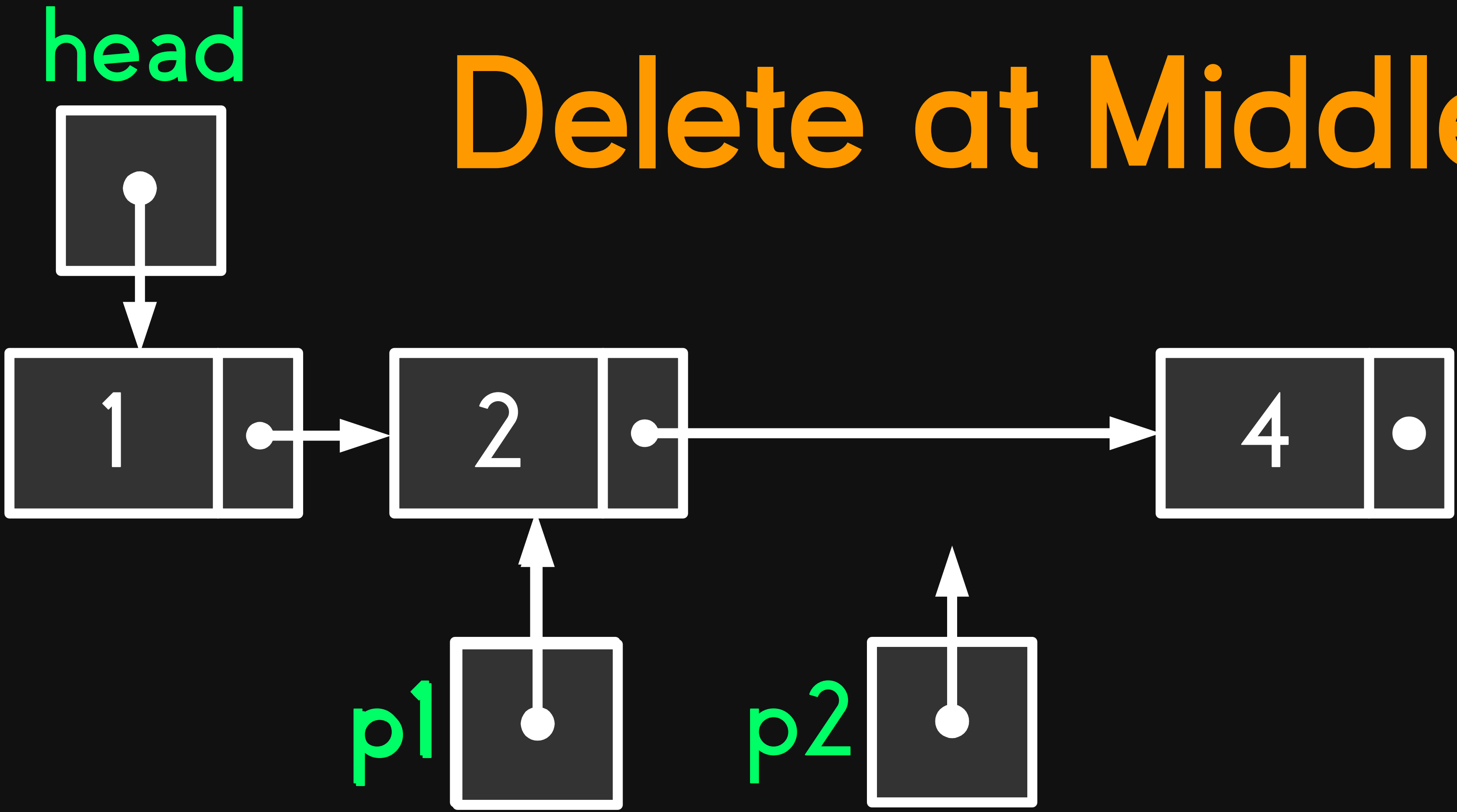
free the node being pointed by p2.

Delete at Middle



free the node being pointed by p2.

Delete at Middle



p2 is now a dangling pointer.

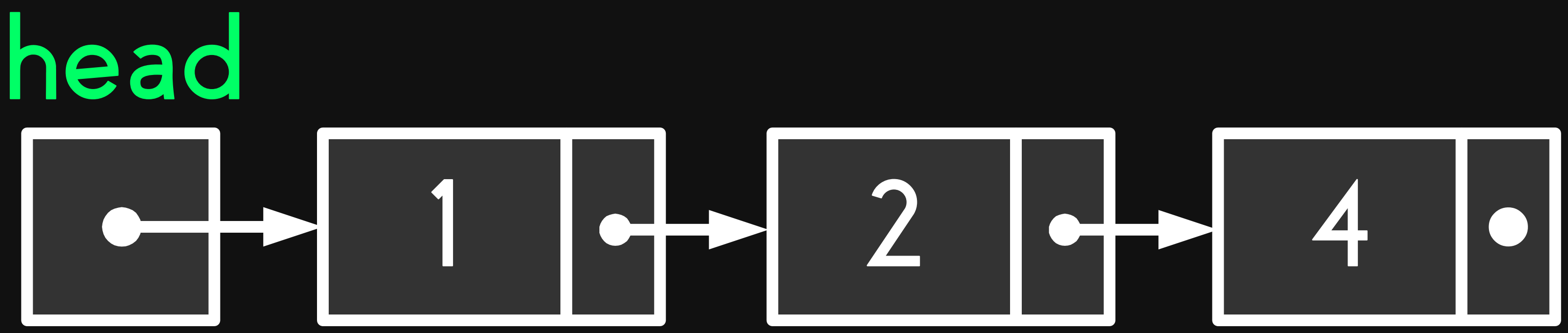
Delete at Tail

delete the
last element
of the list

Delete at Tail

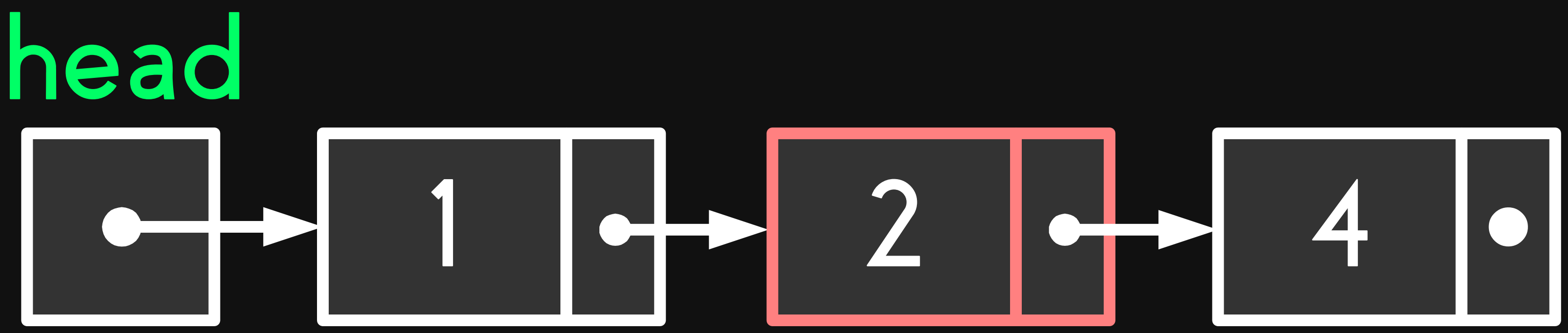
just like in insert,
delete at tail is **a special case of delete at middle**

Delete at Tail



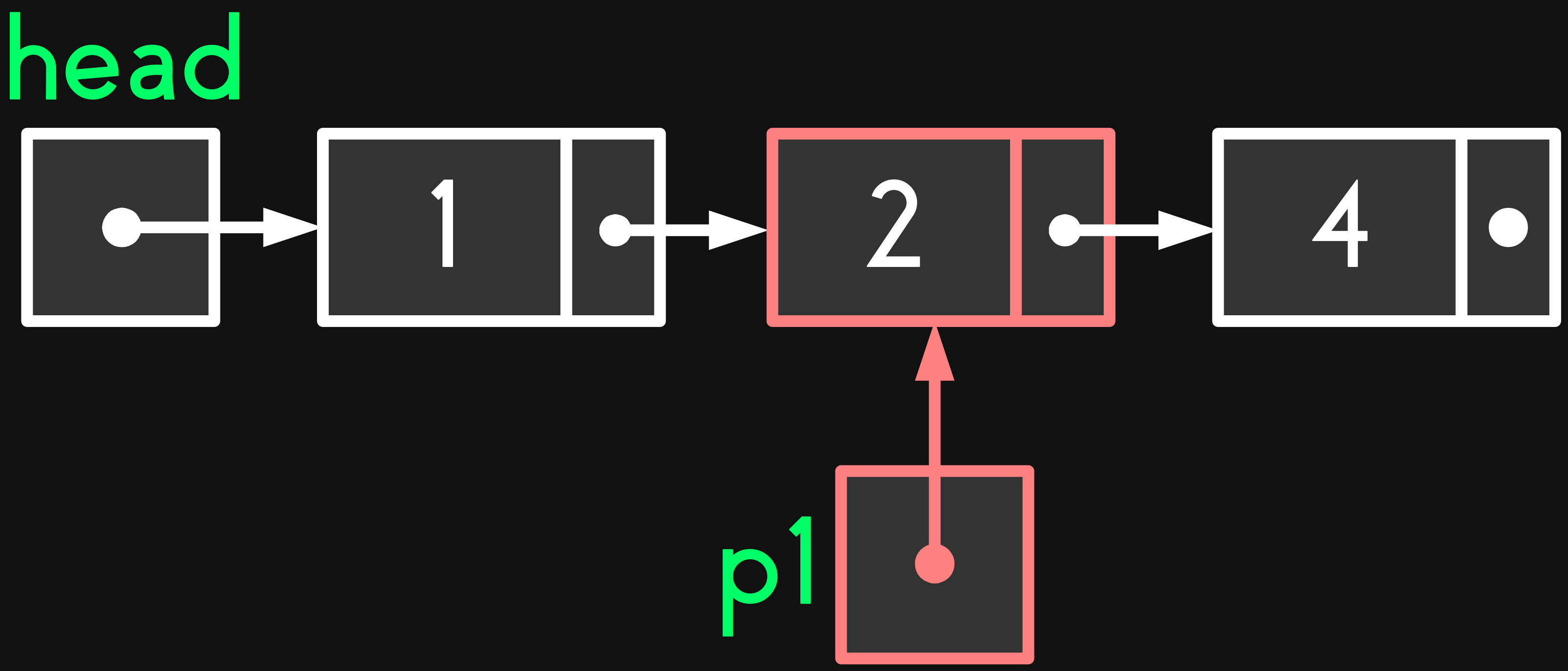
delete the last element (4)

Delete at Tail



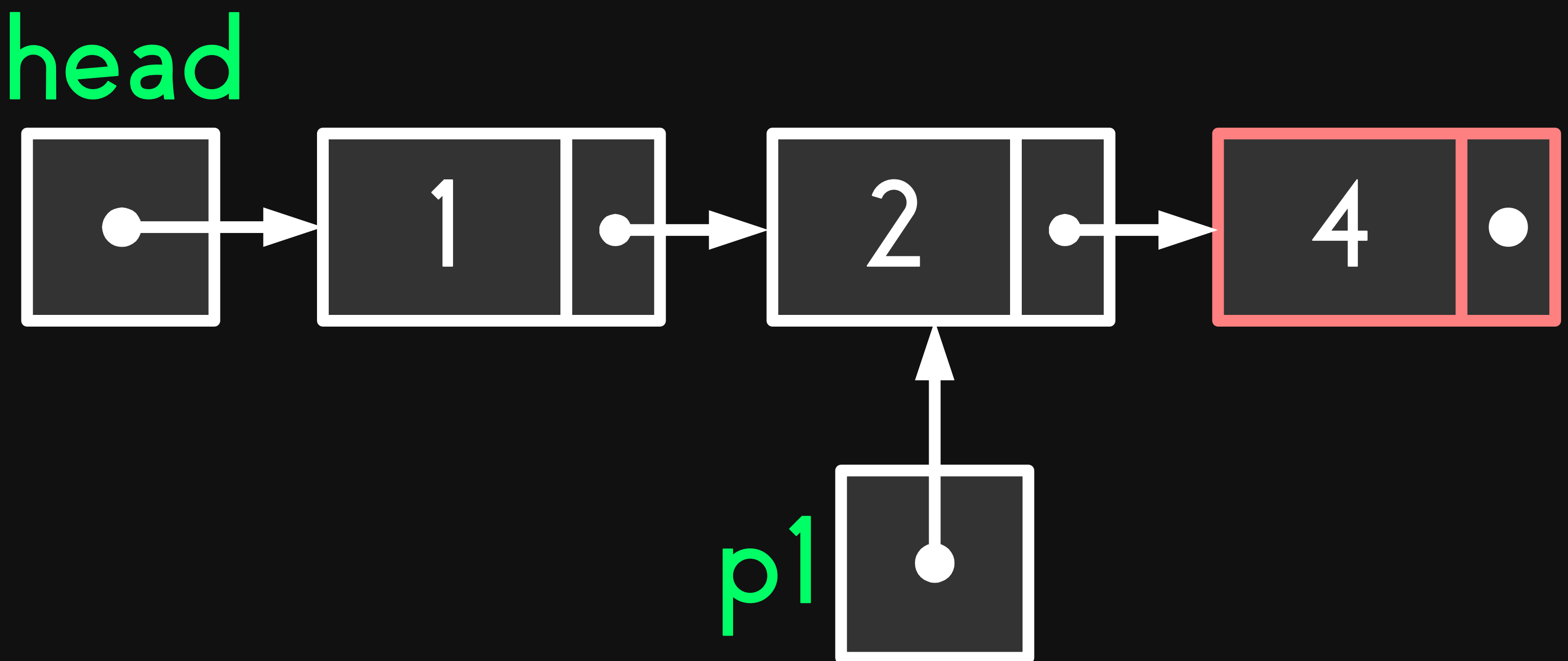
make a pointer (**p1**) point to the node before the tail node

Delete at Tail



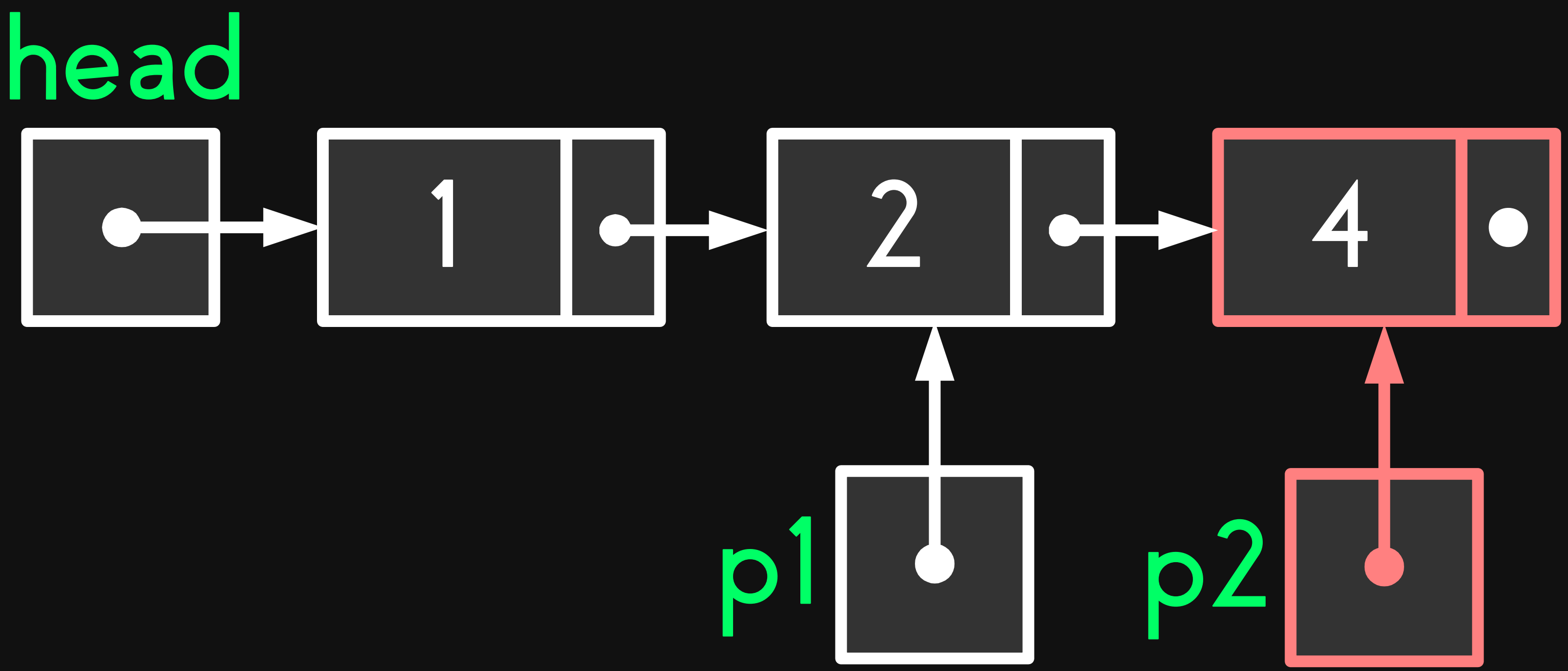
make a pointer (**p1**) point to the
node before the tail node

Delete at Tail



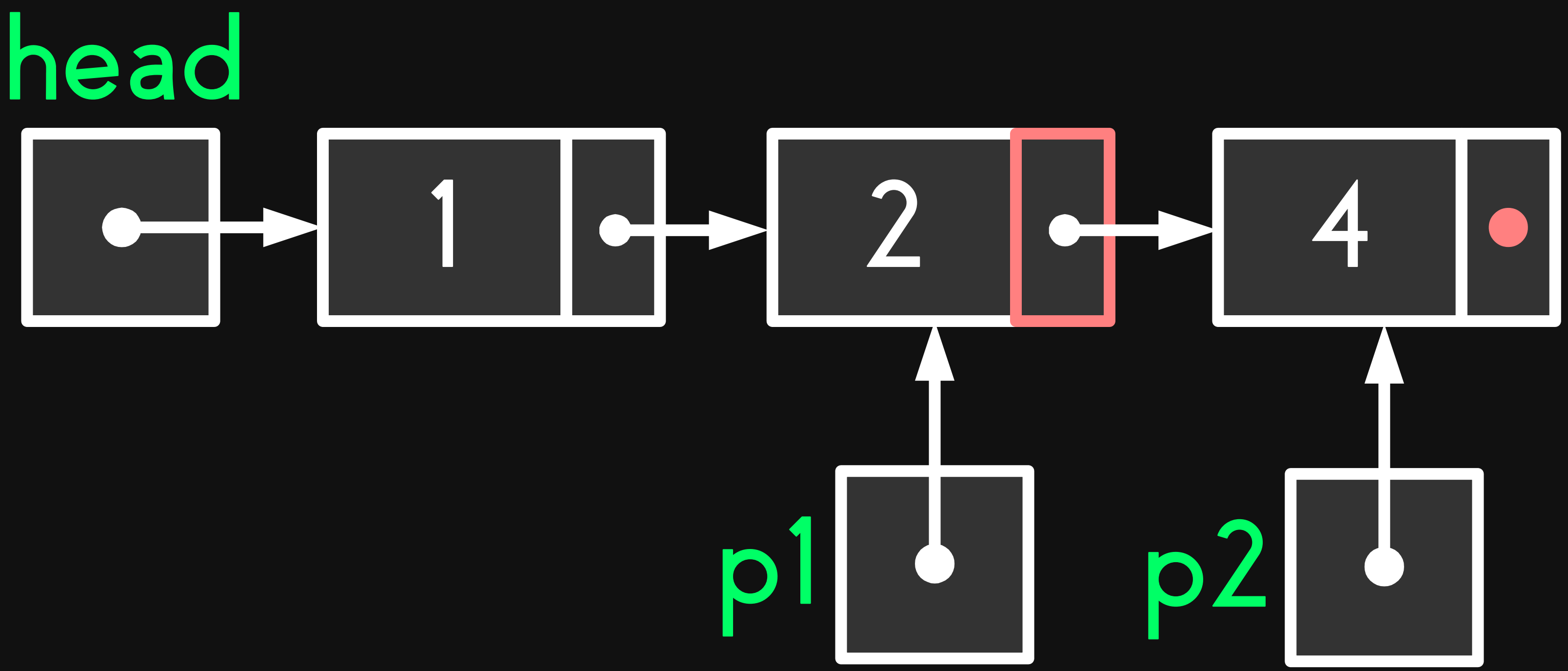
make a pointer (**p2**) point to the last node

Delete at Tail



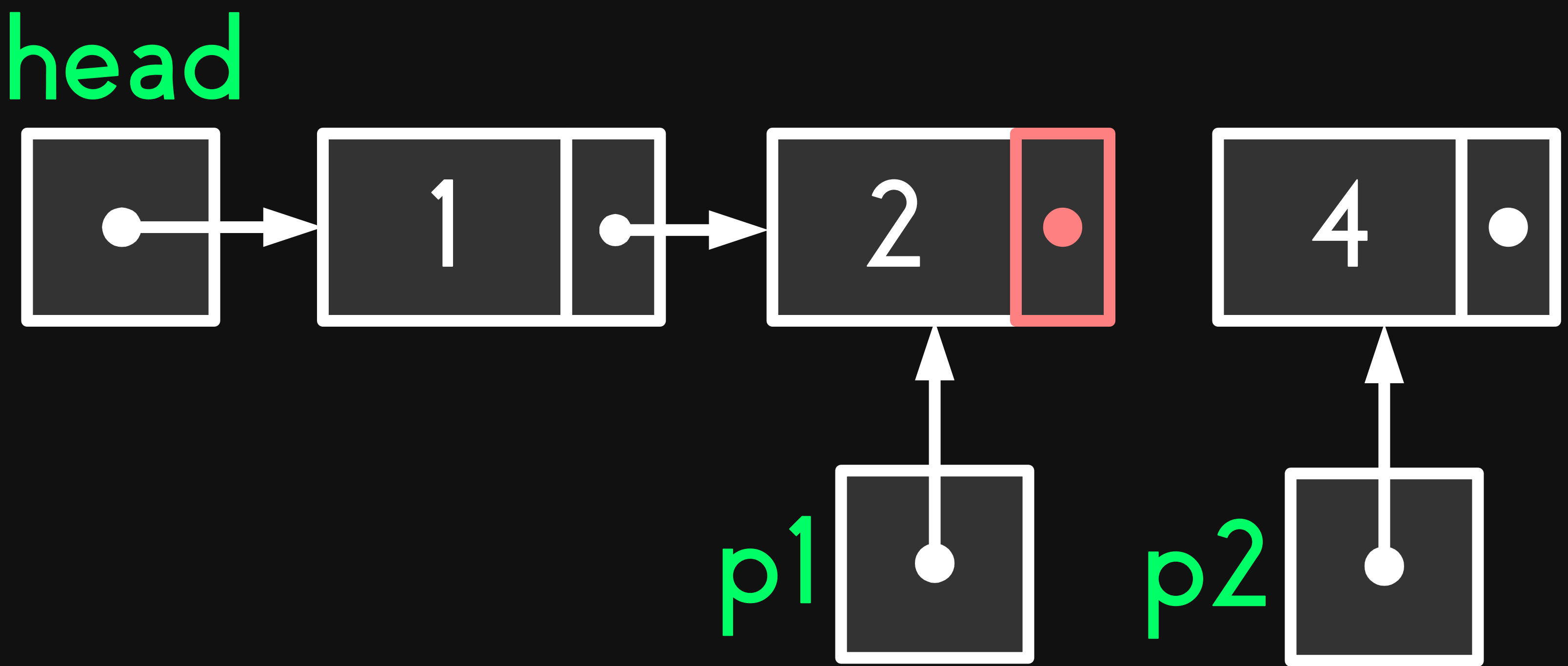
make a pointer (**p2**) point to the last node

Delete at Tail



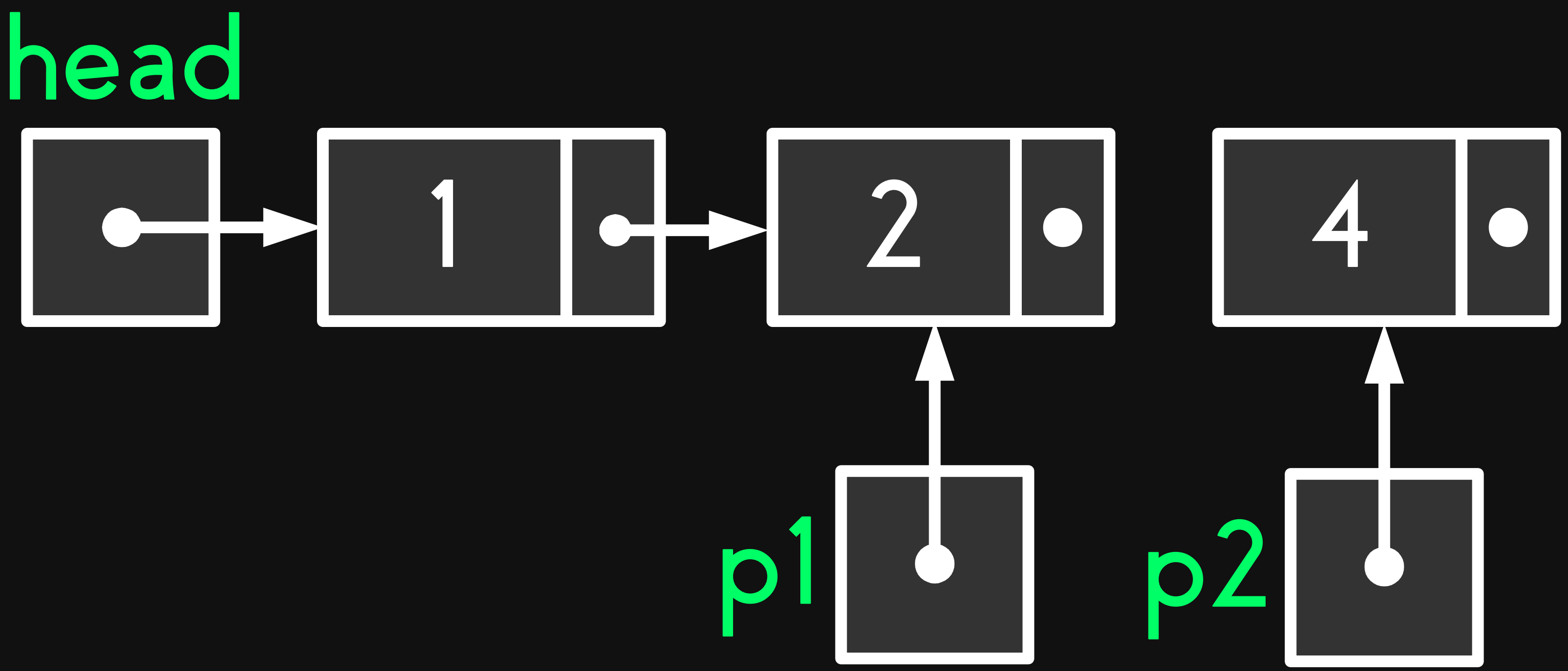
point the next pointer of the node being pointed by **p1** to the node **after** the node to be deleted

Delete at Tail



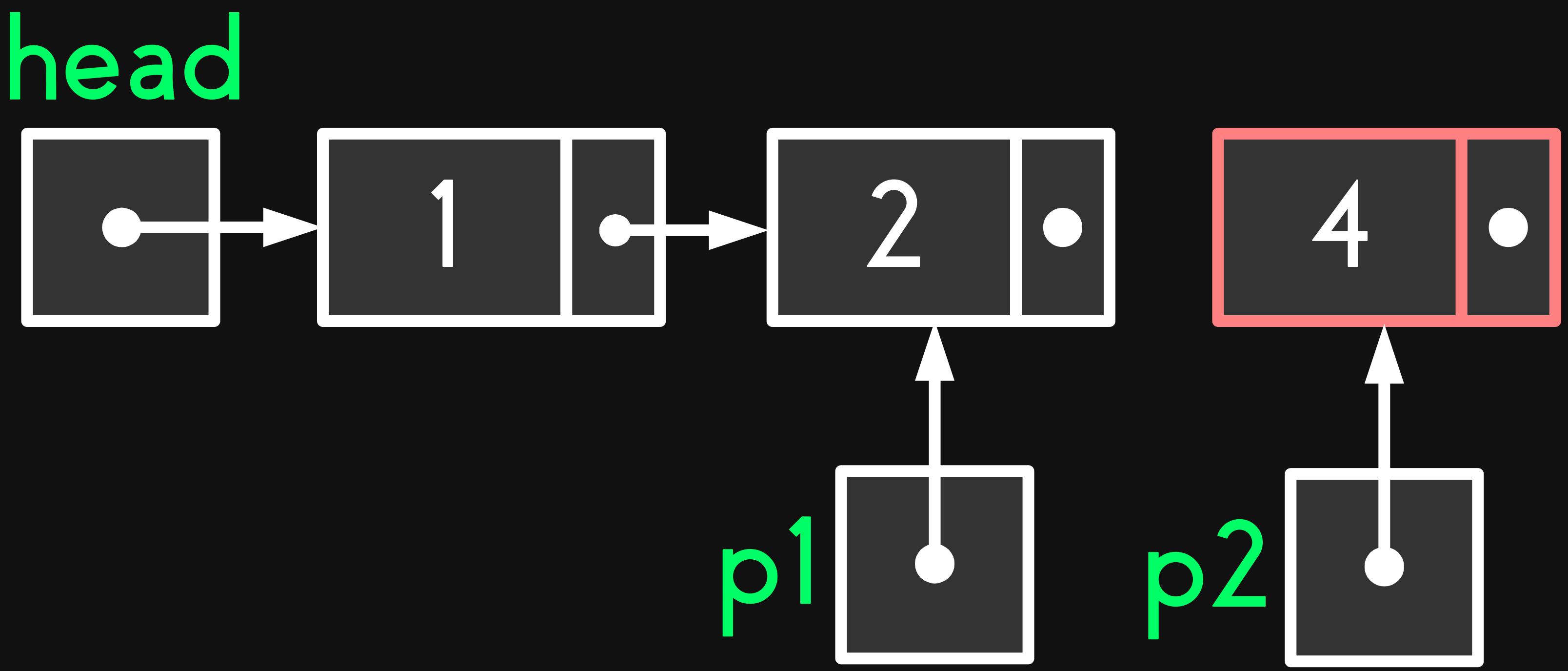
point the next pointer of the node being pointed by **p1** to the node **after** the node to be deleted

Delete at Tail



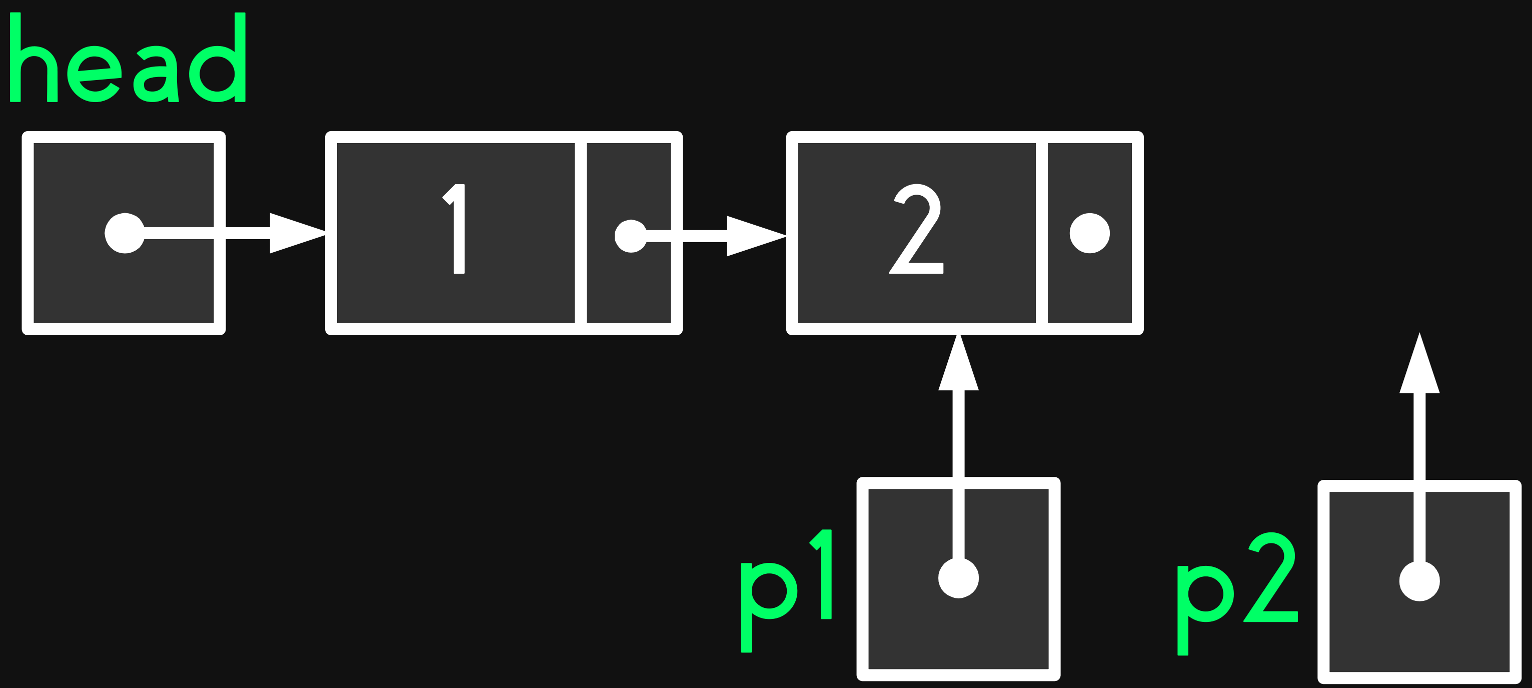
delete the node being pointed by **p2**

Delete at Tail



delete the node being pointed by **p2**

Delete at Tail



p2 is now a dangling pointer.

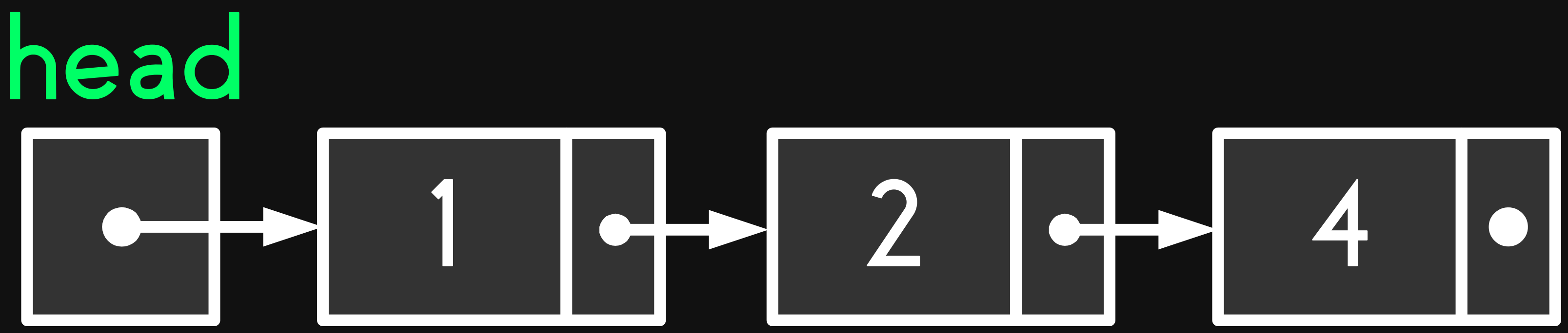
View

prints/shows the
details of the nodes
in a given linked list

View

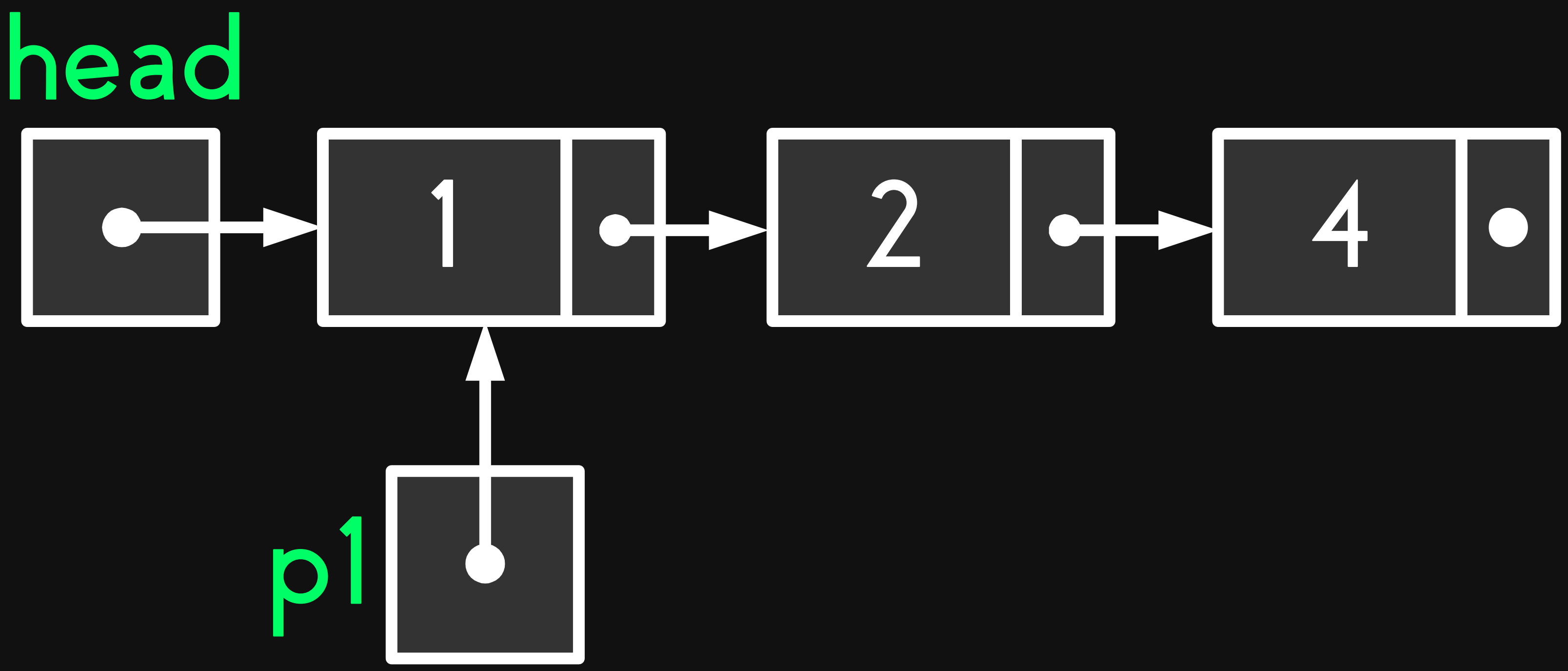
traverses the
linked list

View



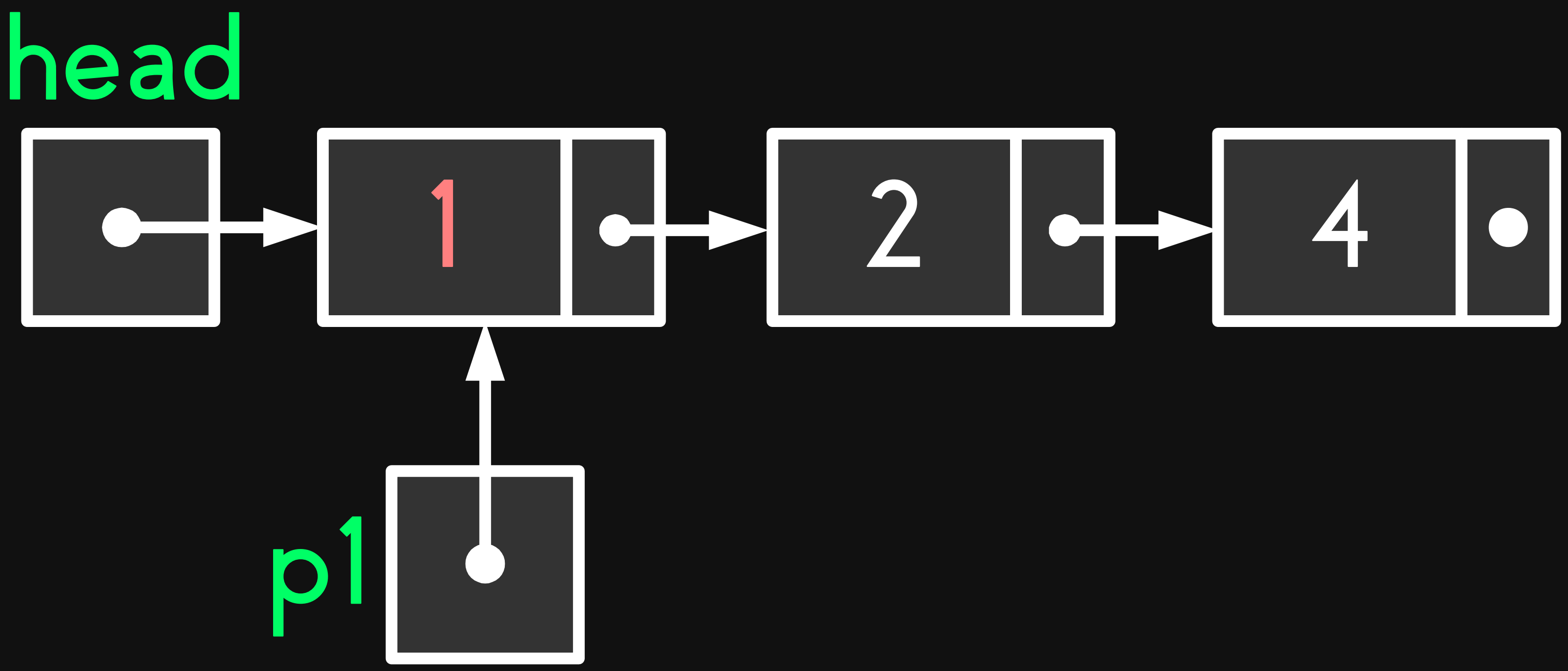
print all the details
of all the nodes

View



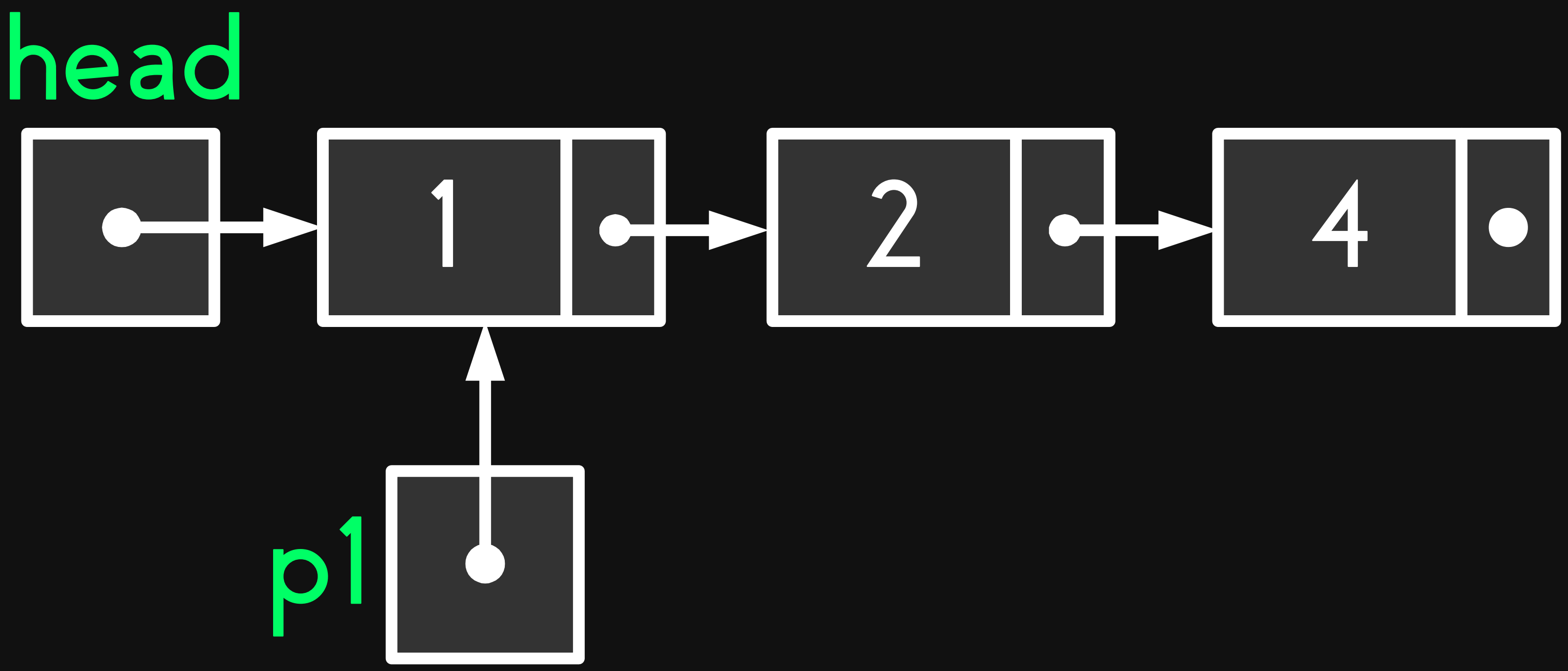
make a pointer point to the
head node

View

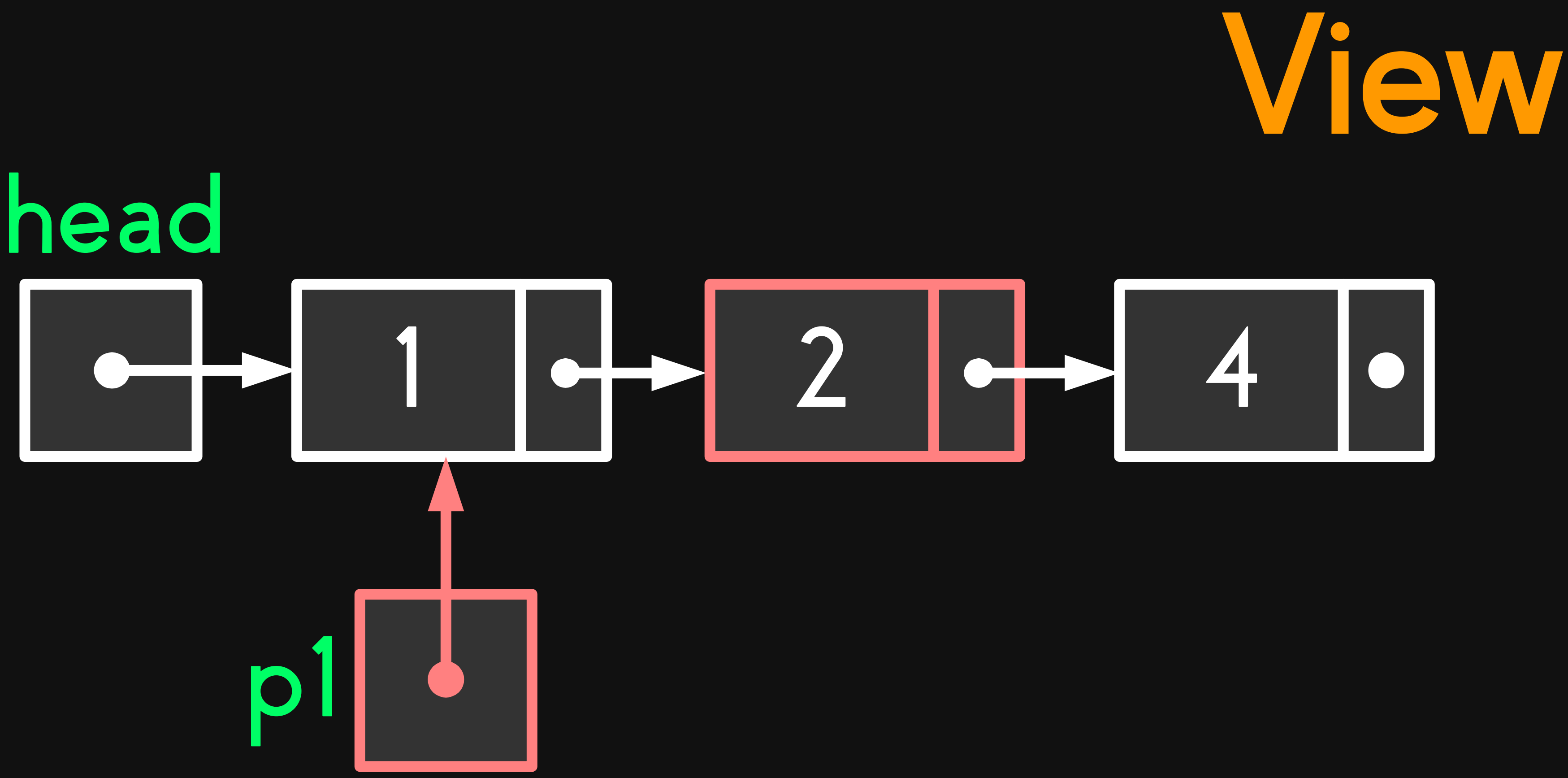


print the data in the node being pointed by **p1**.

View

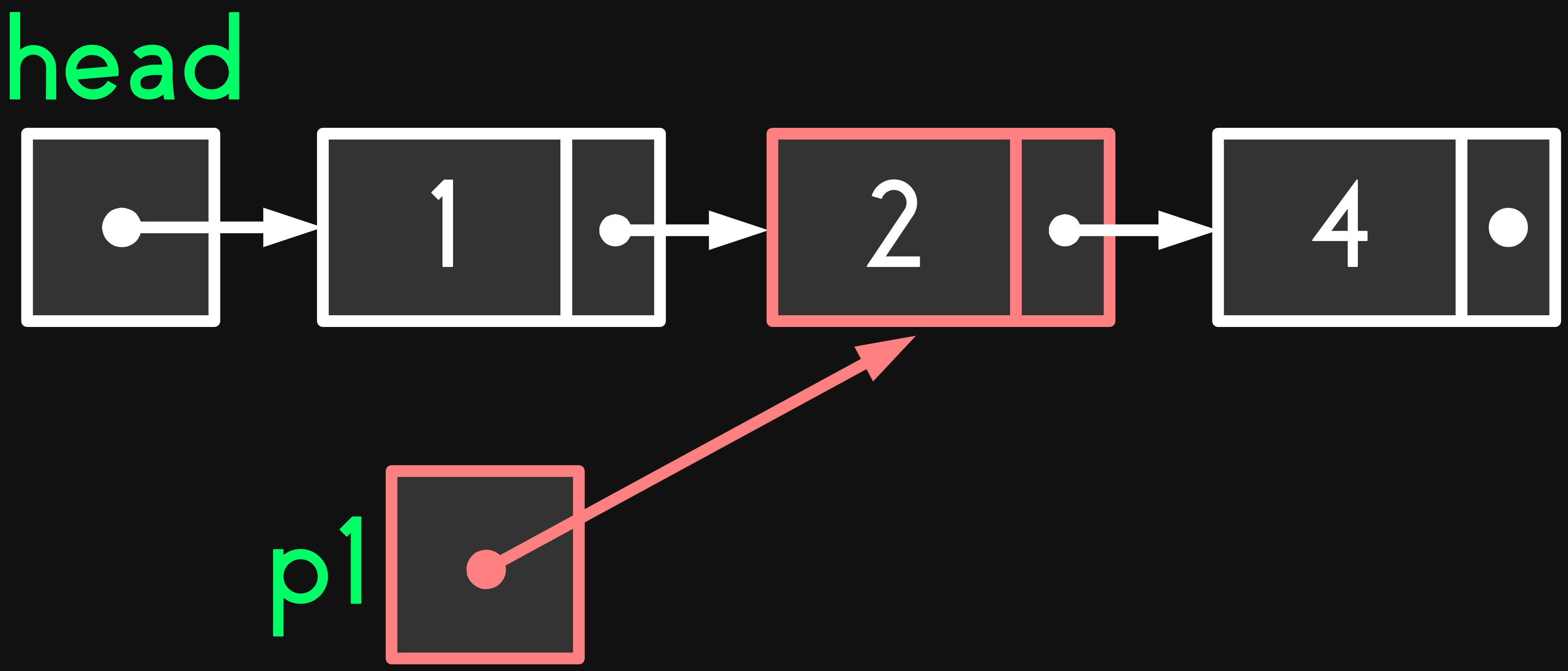


point p1 to the next node.



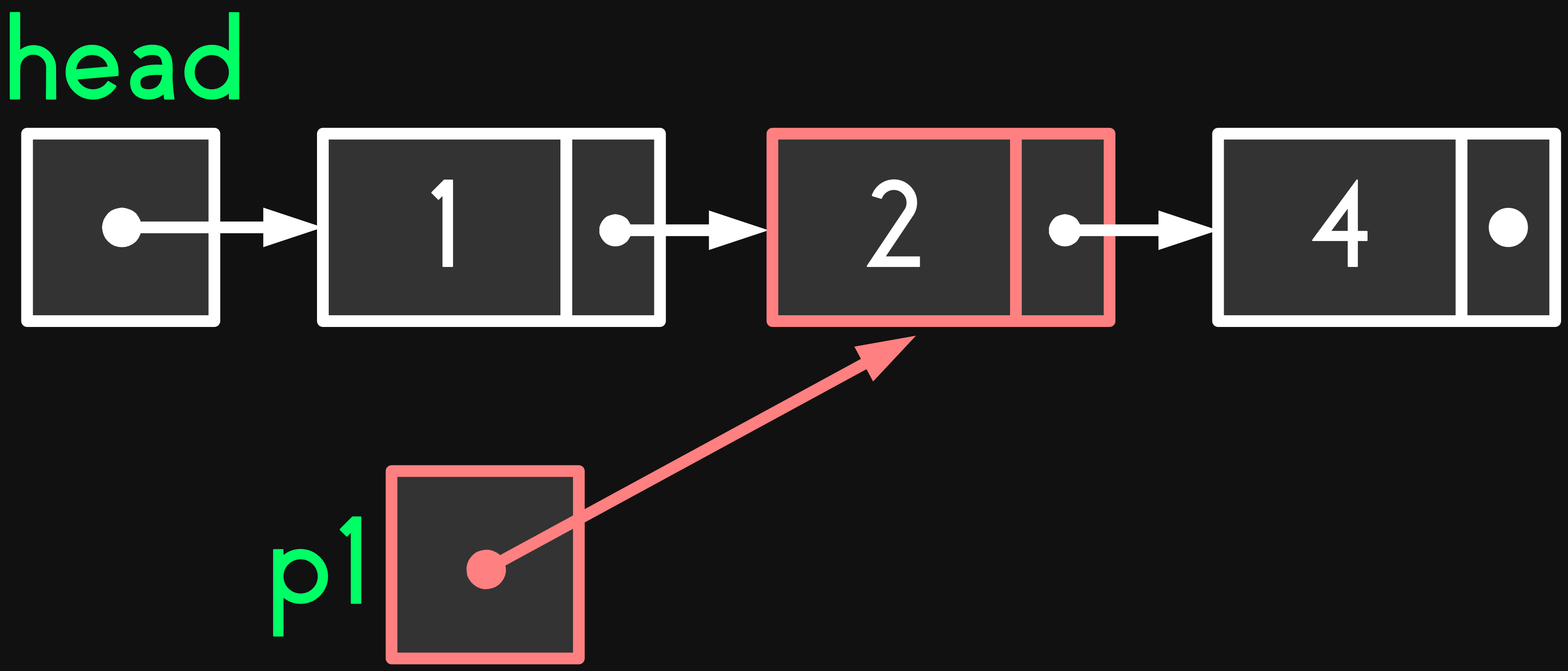
point **p1** to the next node.

View



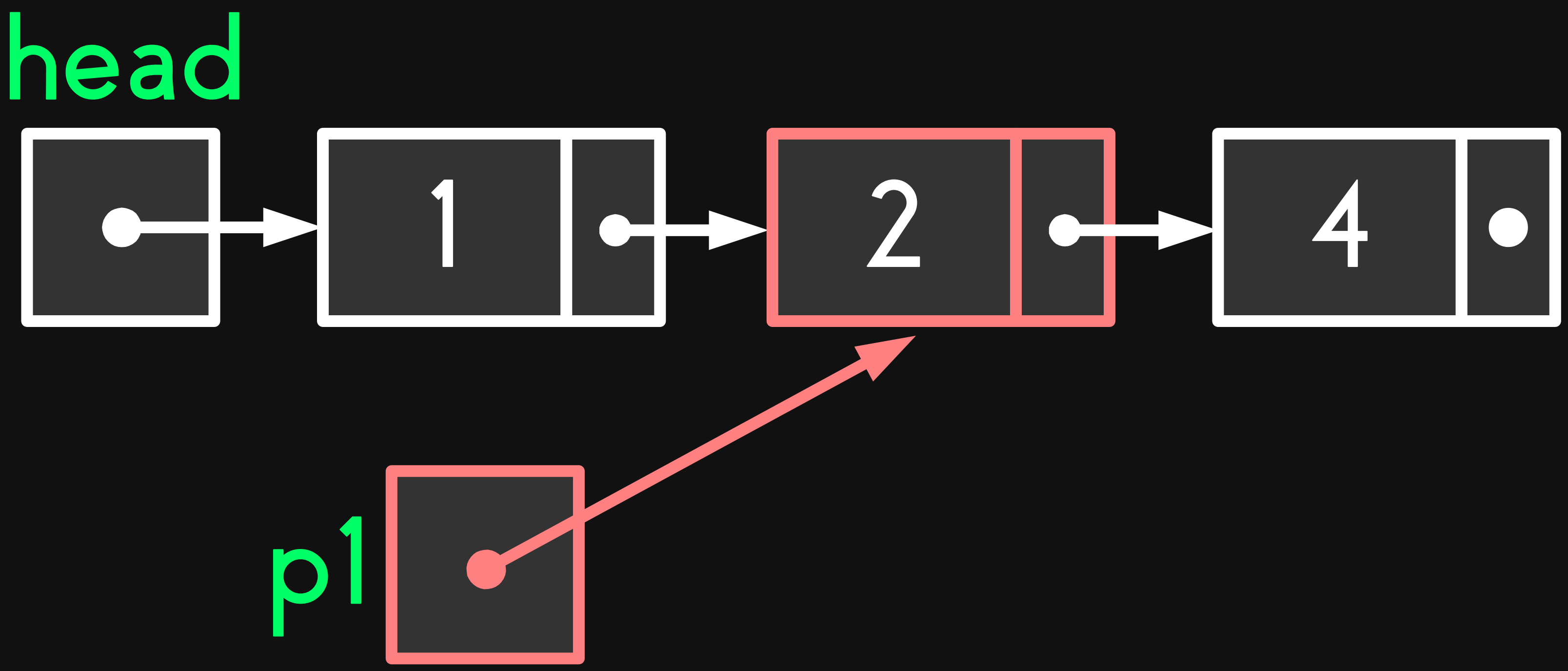
point **p1** to the next node.

View



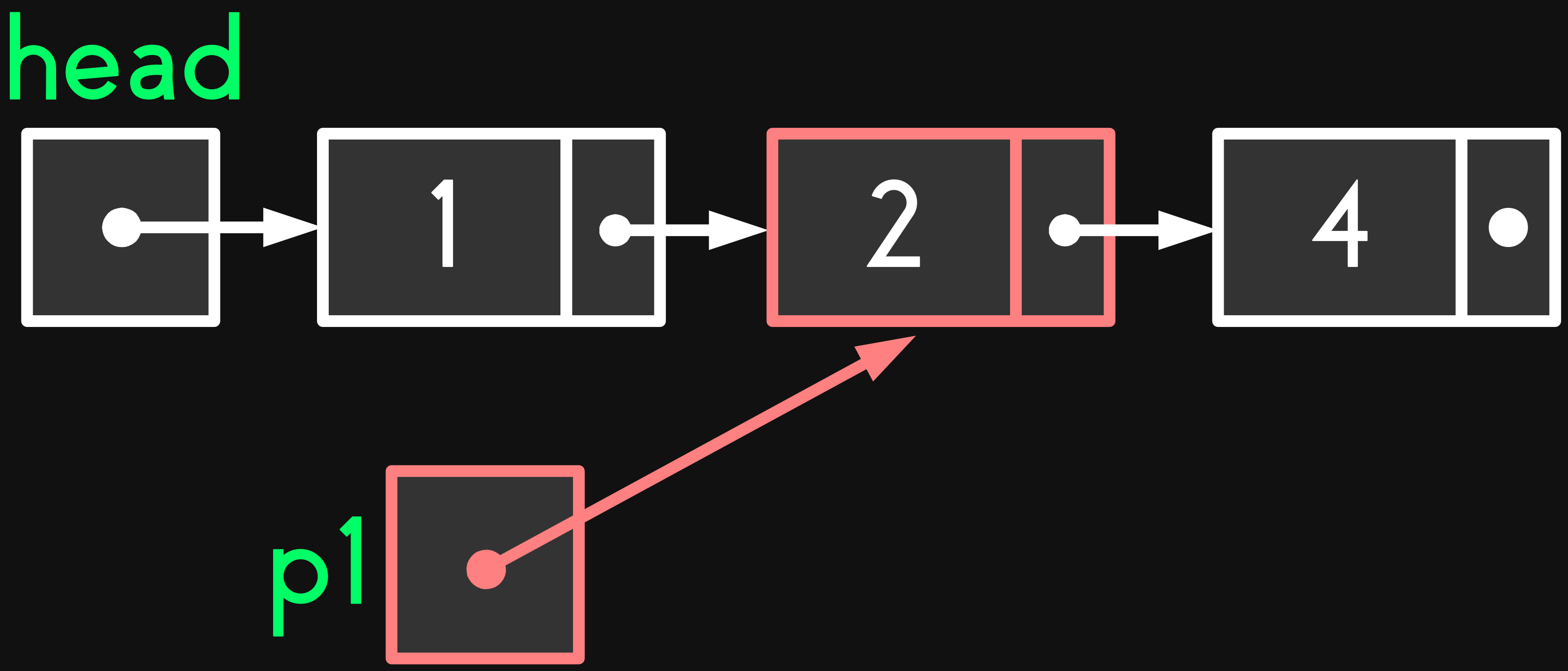
print the data in the node being pointed by **p1**.

View



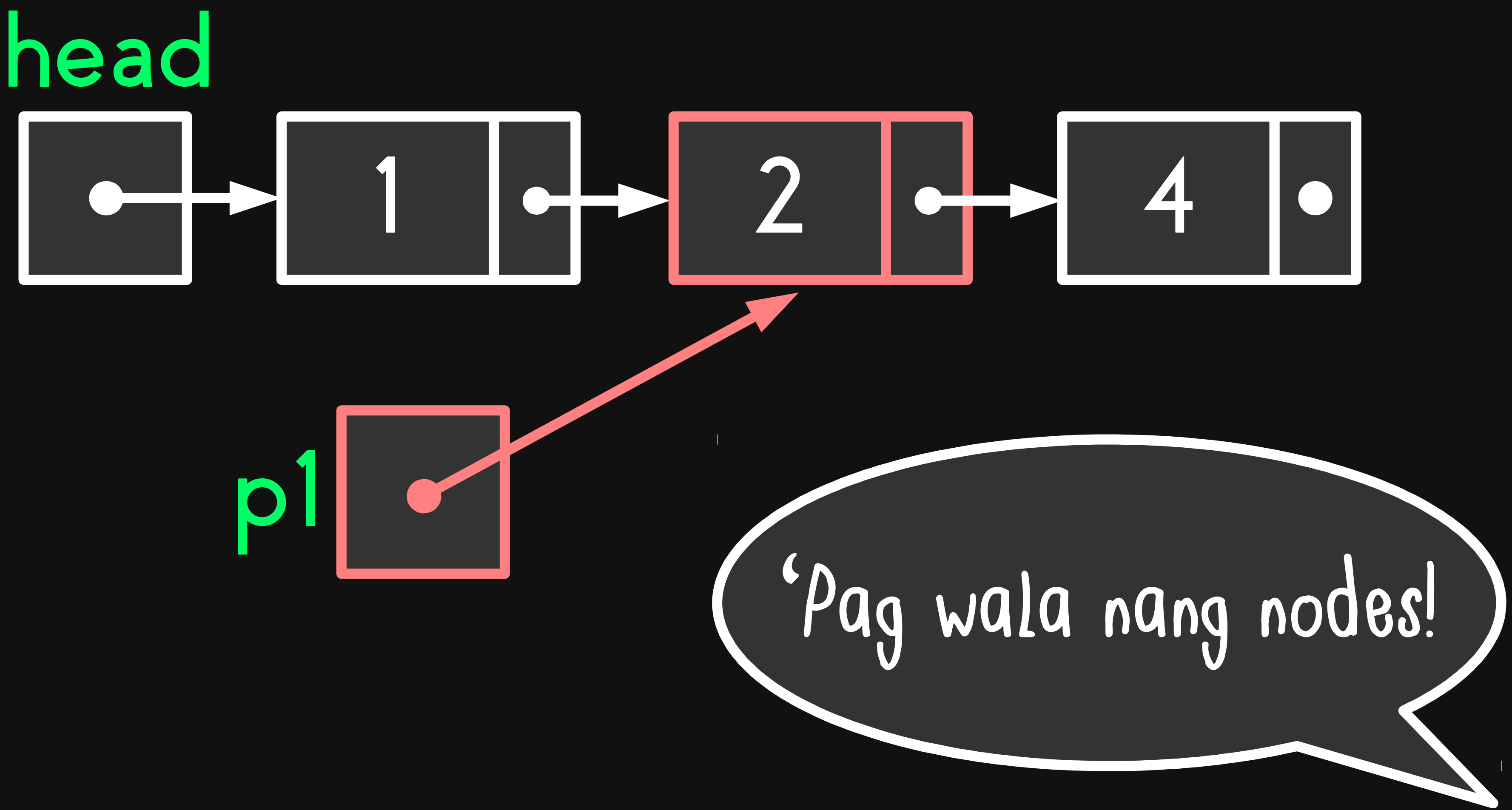
and then move to the next node.
print the data.
move to the next node.
and so on.

View



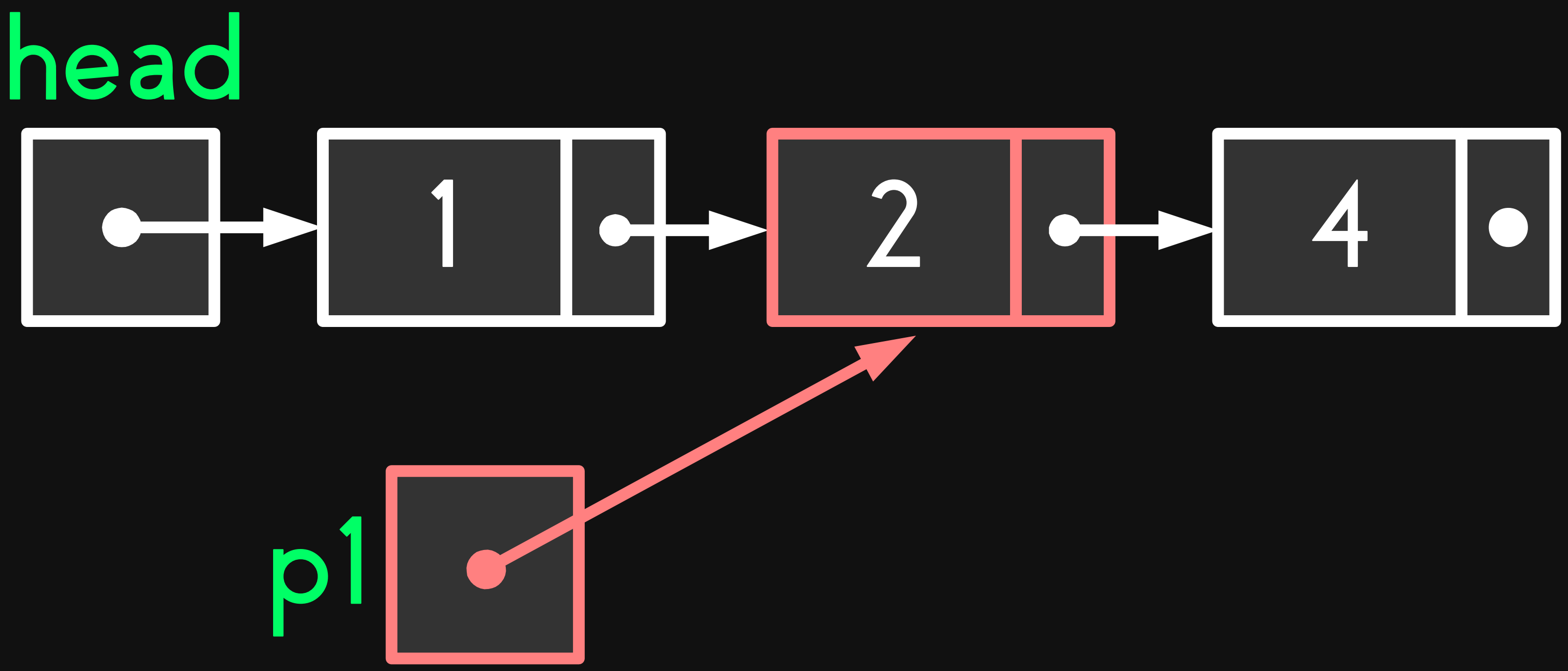
When will this stop?

View



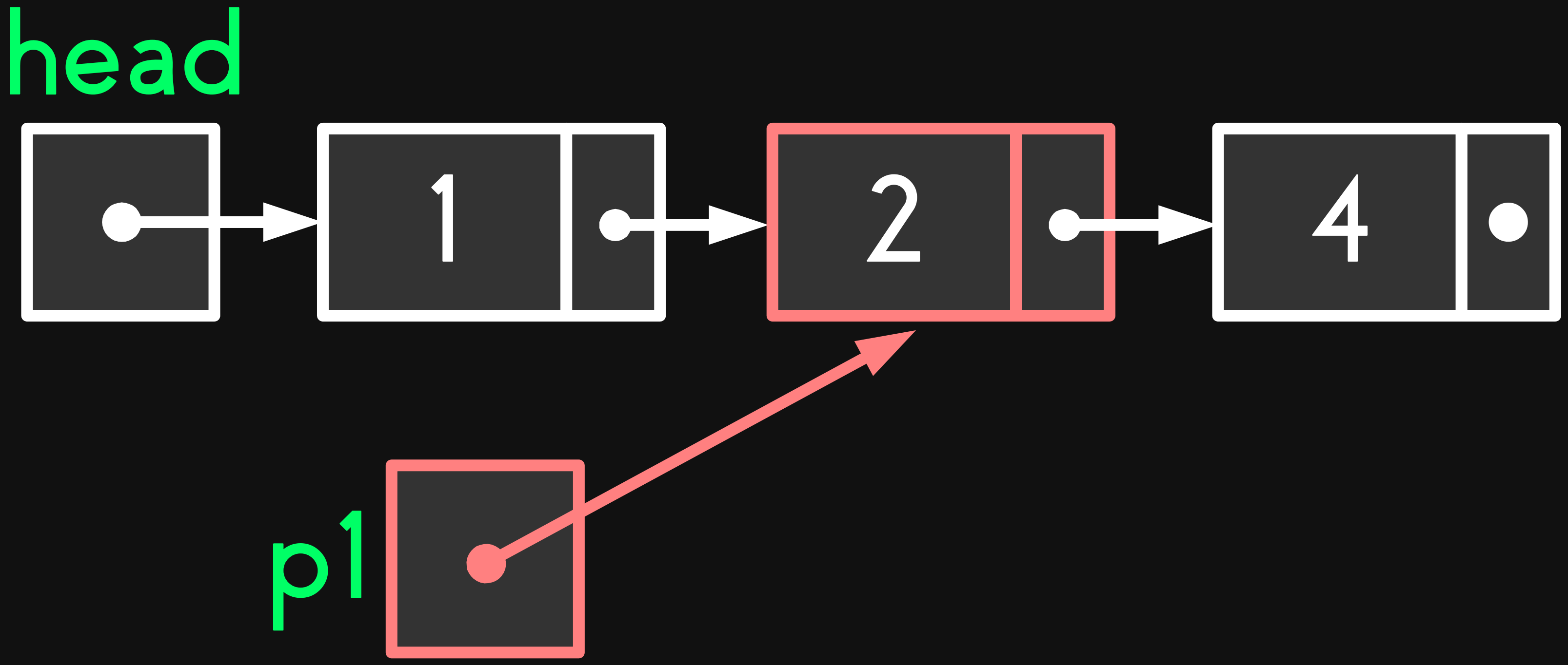
When will this stop?

View



When will you know that there are no more nodes whose data must be printed?

View



HINT: What will be the value of **p1** if there are no more nodes to be printed?

Search

finds a specific item
from the linked list

Search

finds a specific **item**
from the linked list

similar to the **view** operation

Search

stops once
the item is found
or the end of the list
is reached

LINKED LISTS VS ARRAYS

Linked Lists vs Arrays

linked lists
save memory

Linked Lists vs Arrays

allocated **memory will NEVER exceed what is needed** by the program

Linked Lists vs Arrays

dynamic **arrays can handle the change in maximum size** but there is **a possibility that there will be unused allocated memory.**