Lecture 6 - Linked Lists

Foundations of Computer Science (4CC505)

Dr Sam O'Neill

Disclaimer

There are a lot of slides.

However, most are pictures and animations!

Visualgo.net

Please use the following to explore linked lists. We will use this more throughout the module.

Linked List (Single, Doubly), Stack, Queue, Deque – VisuAlgo

Sequence Interface

- Maintains a sequence of n items, e.g. `34,25,35,54` or `"sam", "joe", 1`
- Supports sequence operations

e.g. Python List

Sequence Operations

Note that this is not written as Python, we are just describing the type of things (operations) we should be able to do to the sequence.

Name	Description
`create(X)`	create sequence from items in `X`
`size()`	return the length of the sequence
`get(i)`	return the item at index `i`
`set(i,x)`	replace the item at index `i` with `x`
`insert(i,x)`	add `x` to position `i` (this will move all previous items at index `i`, `i+1`,... etc up 1)
`delete(i)`	delete the item at index `i` (this will move all previous items at index `i`, `i+1`,... etc down 1)

Example: Python List

```
demo_list = [2,5,1,66,3,4,23,42]

len(demo_list)            # size() - return the length of the sequence

demo_list[2]              # get(2) - return the item at index 2

demo_list[2] = 34         # set(2,34) - replace the item at index 2 with 34

demo_list.insert(2,999) # insert(2,999) - add 999 to index 2

demo_list.pop(4)          # delete(4) - delete the item at index 4
```

Additional Sequence Operations

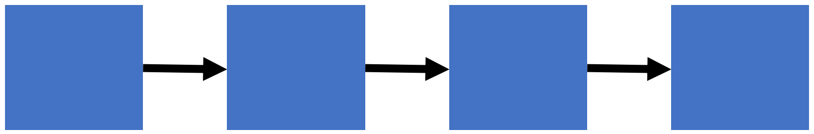
We will also consider the following operations.

Name	Description
`insert_first(x)`	add `x` as the first item. Same as `insert(0,x)`
`delete_first()`	delete the first item. Same as `delete(0)`
`insert_last(x)`	add `x` as the last item. Same as `insert(size(),x)`
`delete_last()`	delete the last item. Same as `delete(size()-1)`

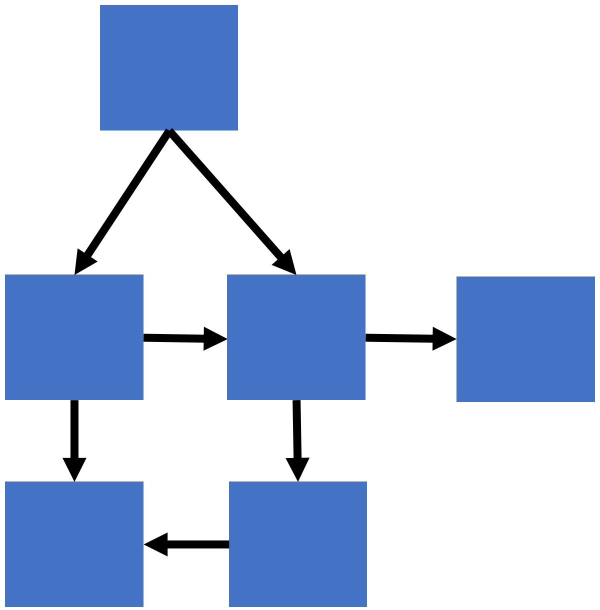
Example: Python List

```
demo_list = [2,5,1,66,3,4,23,42]

demo_list.insert(0,99)    # insert_first(99) - add 99 as the first item

demo_list.pop(0)          # delete(0) - delete the first item

demo_list.append(999)   # insert_last(999) add 999 as the last item

demo_list.pop()           # delete_last() - delete the last item
```

Linear ‘Vs’ Nonlinear Data Structures

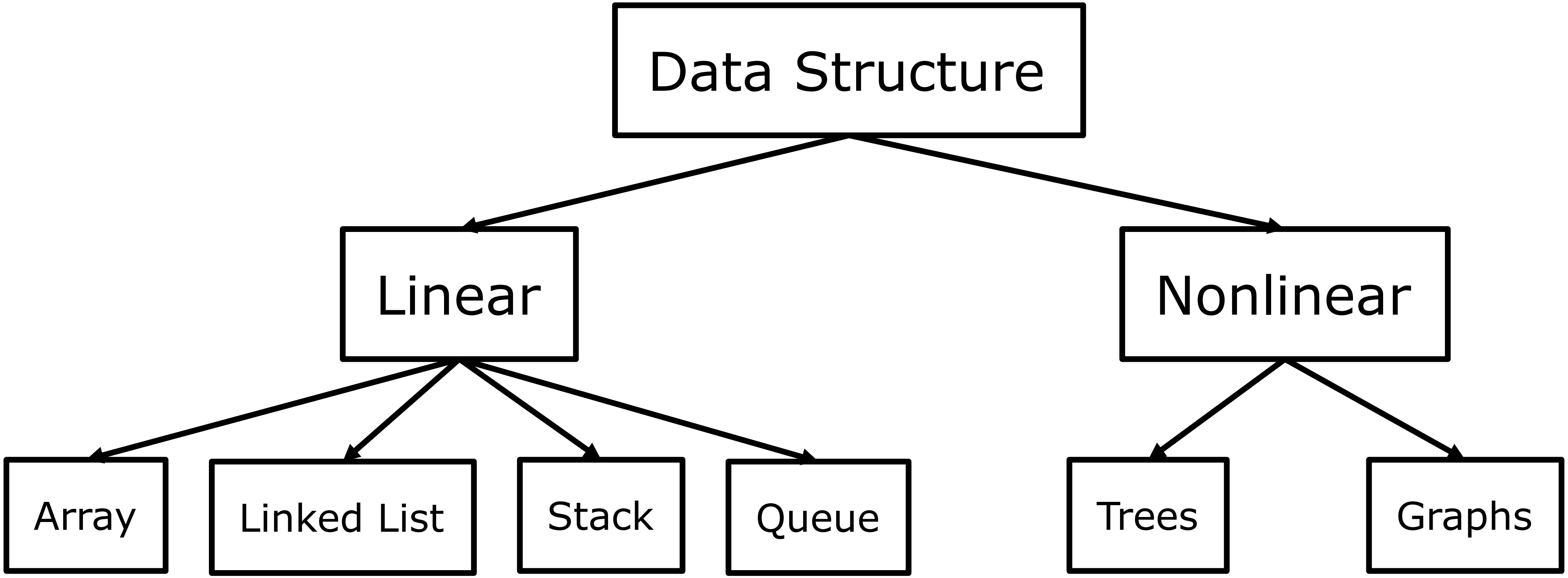


Linear



Nonlinear

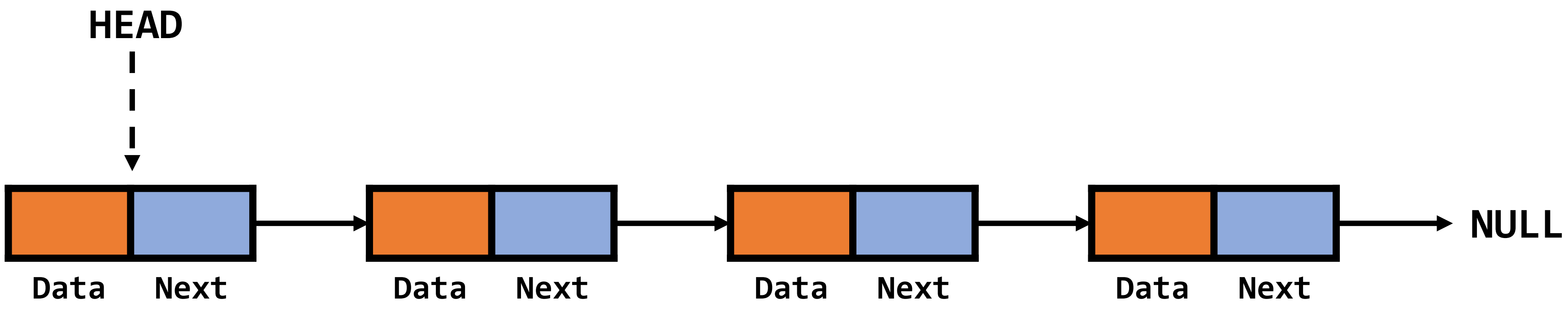
Linear ‘Vs’ Nonlinear Data Structures



Singly Linked List

Demo

Singly Linked List



- A sequence of nodes that are linked together
- We have access to the **HEAD** (first) node
- Each node points to the next node
- Last node doesn't point to anything (**NULL**)

Motivation

- Arrays and Dynamic arrays are slow for insertions

- We have to resize for insertions, this means an expensive copy $O(n)$

- Sometimes we don't know the space we require (wastage)

Applications

1. Implementation of Stacks
2. Implementation of Graphs
3. Dynamic Memory Allocation

I could have listed more, but these 3 are vitally important in computer science.

We will see why in upcoming lectures.

Python Code

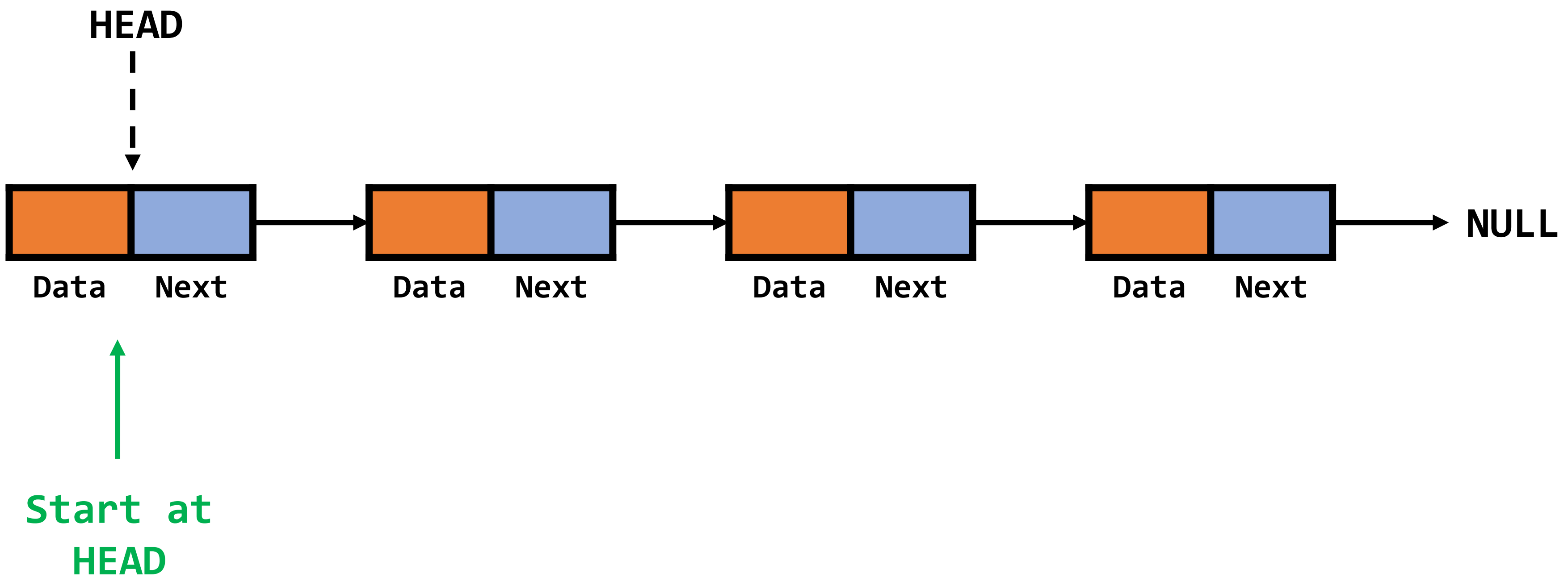
```
# Linked list implementation in Python
# Programiz https://www.programiz.com/dsa/linked-list

class Node:
  """ Represents a single node"""
  def __init__(self, item):
      self.item = item
      self.next = None

class LinkedList:
  """ The whole linked list"""
  def __init__(self):
      self.head = None

if __name__ == '__main__':

  linked_list = LinkedList()

  # Assign item values
  linked_list.head = Node(1)
  second = Node(2)
  third = Node(3)

  # Connect nodes
  linked_list.head.next = second
  second.next = third

  # list will not contain 1 -> 2 -> 3
```

You can find a full implementation in the module folder that supports all operations.

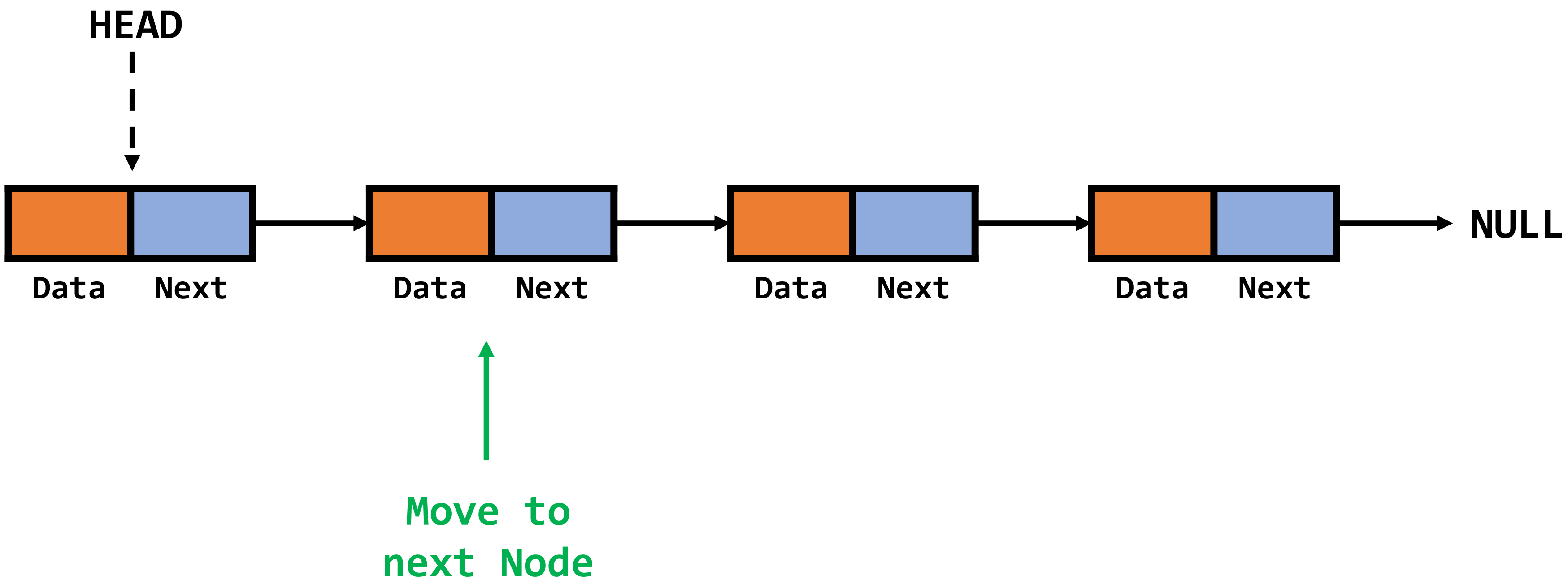
Traverse a Singly Linked List

We can traverse (visit every node) a singly linked list in $O(n)$

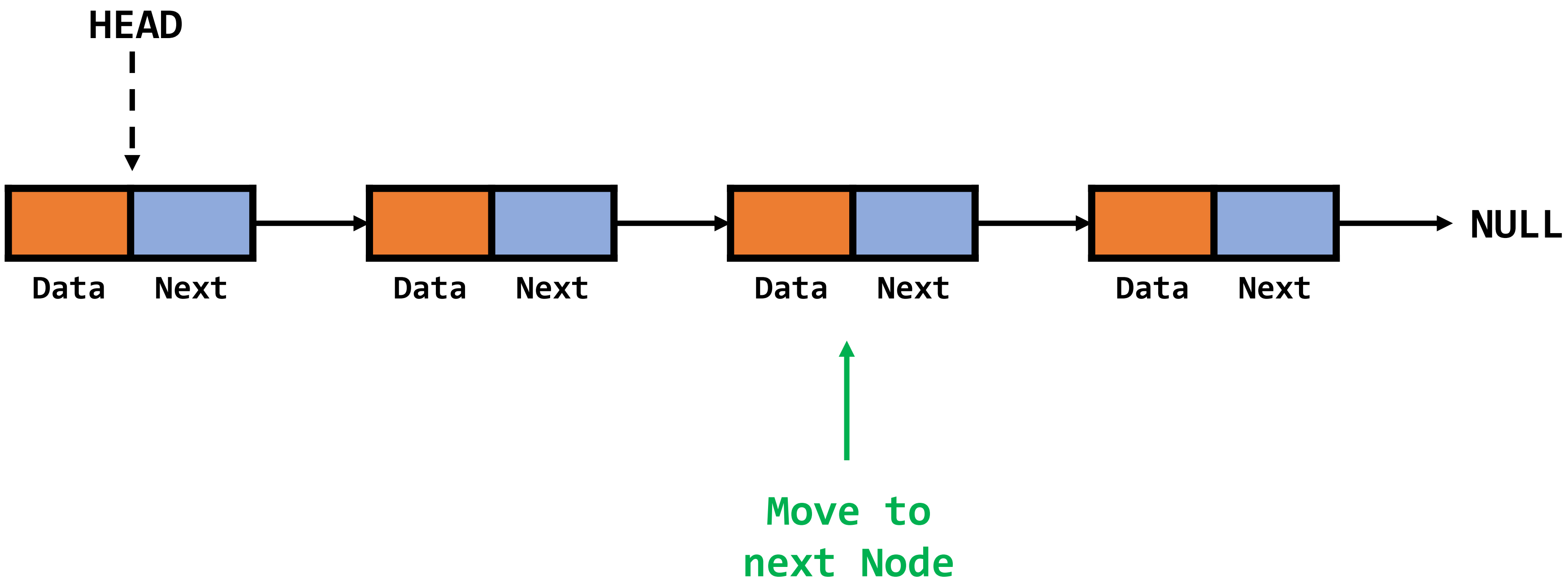
Traverse a Singly Linked List



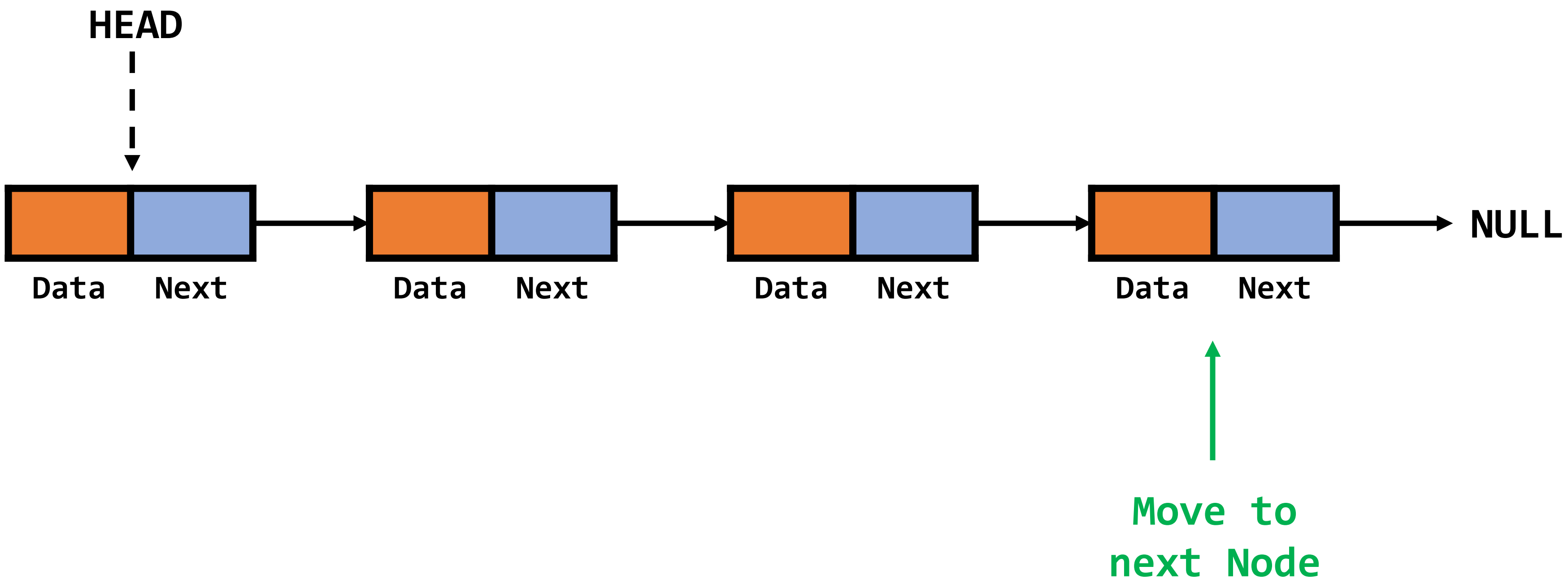
Traverse a Singly Linked List



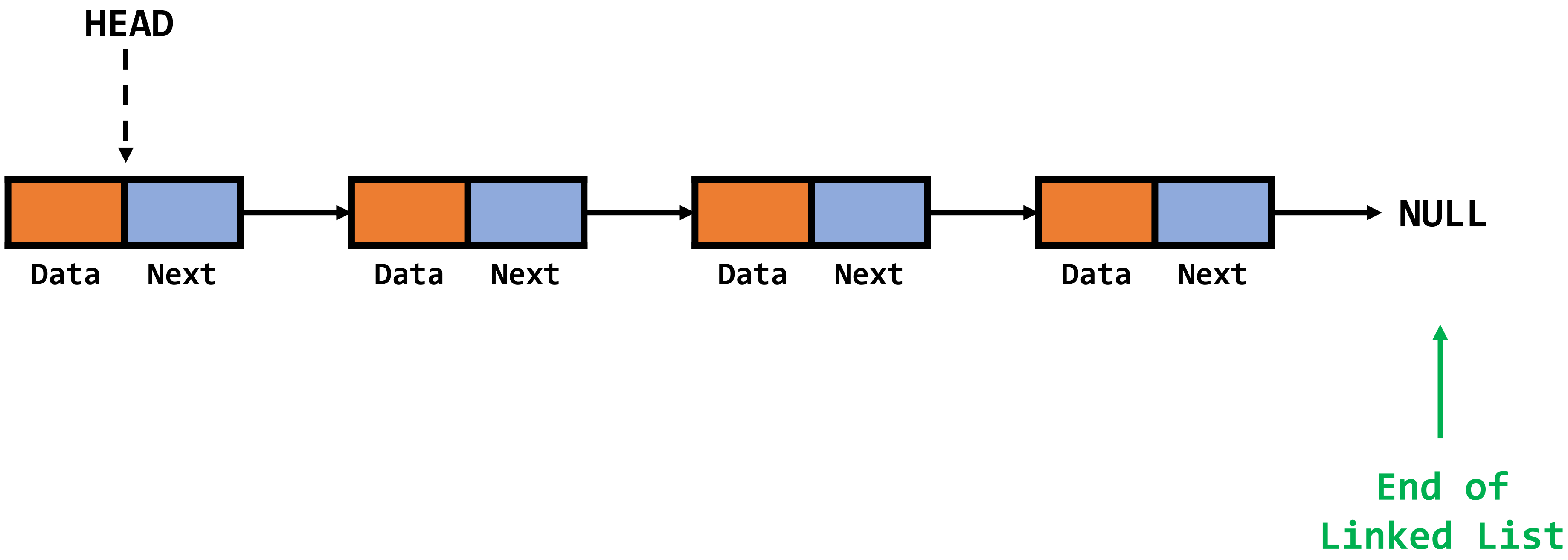
Traverse a Singly Linked List



Traverse a Singly Linked List



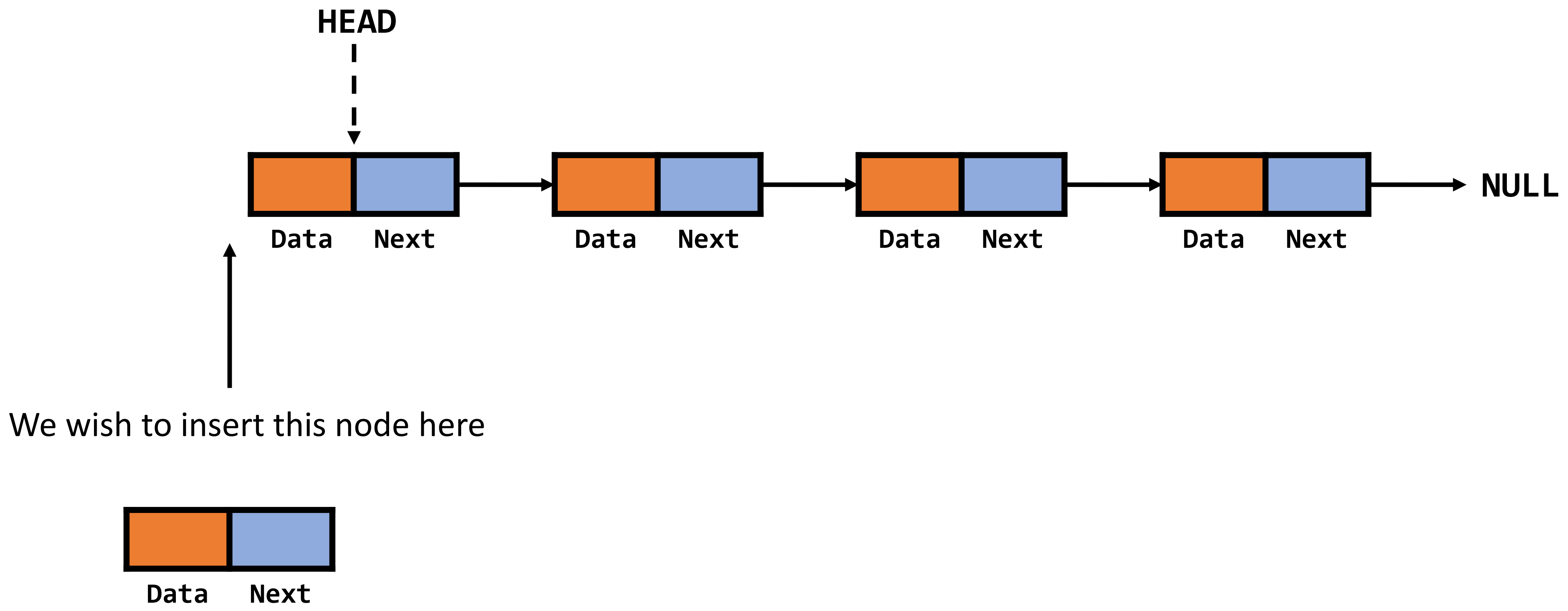
Traverse a Singly Linked List



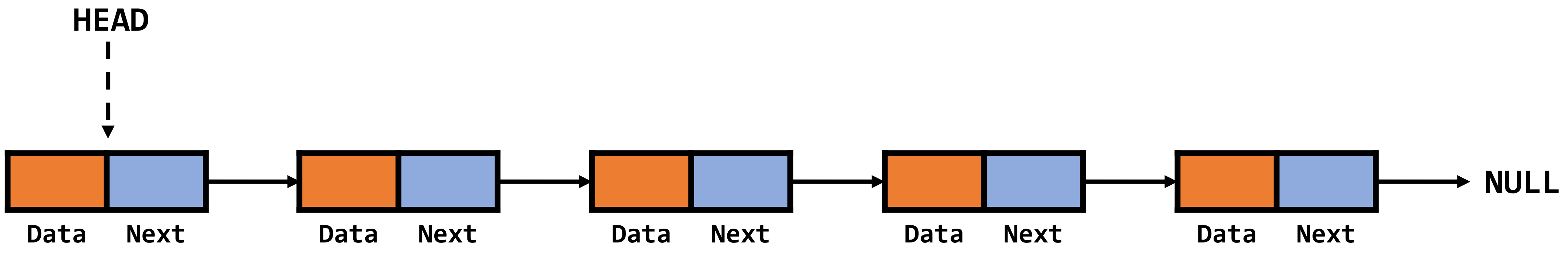
insert_first()

Insert at the beginning of a singly linked list

insert_first()

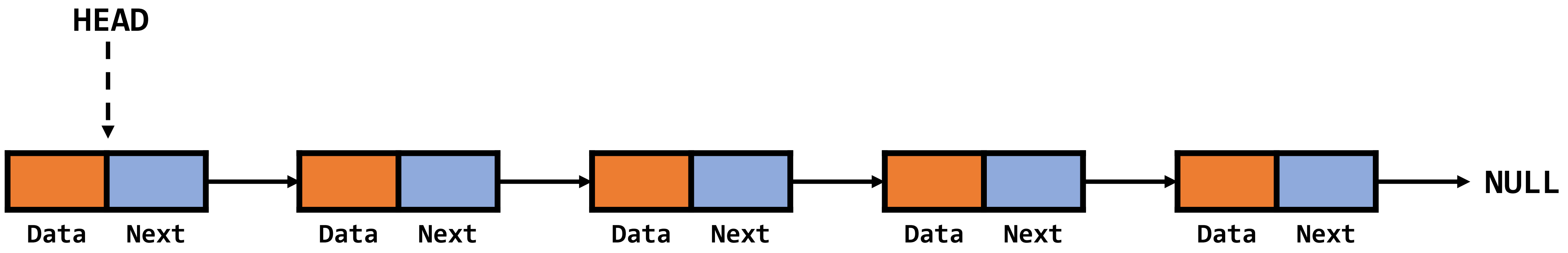


`insert_first()`



This costs $O(1)$. Why?

`insert_first()`



This costs $O(1)$. Why?

We are storing the **HEAD** of the linked list, which means we know what the first node is so we can just insert it.

We reassign the new node as the **HEAD** and set its **next** to be the previous **HEAD**.

Python Code - With `insert_first()`

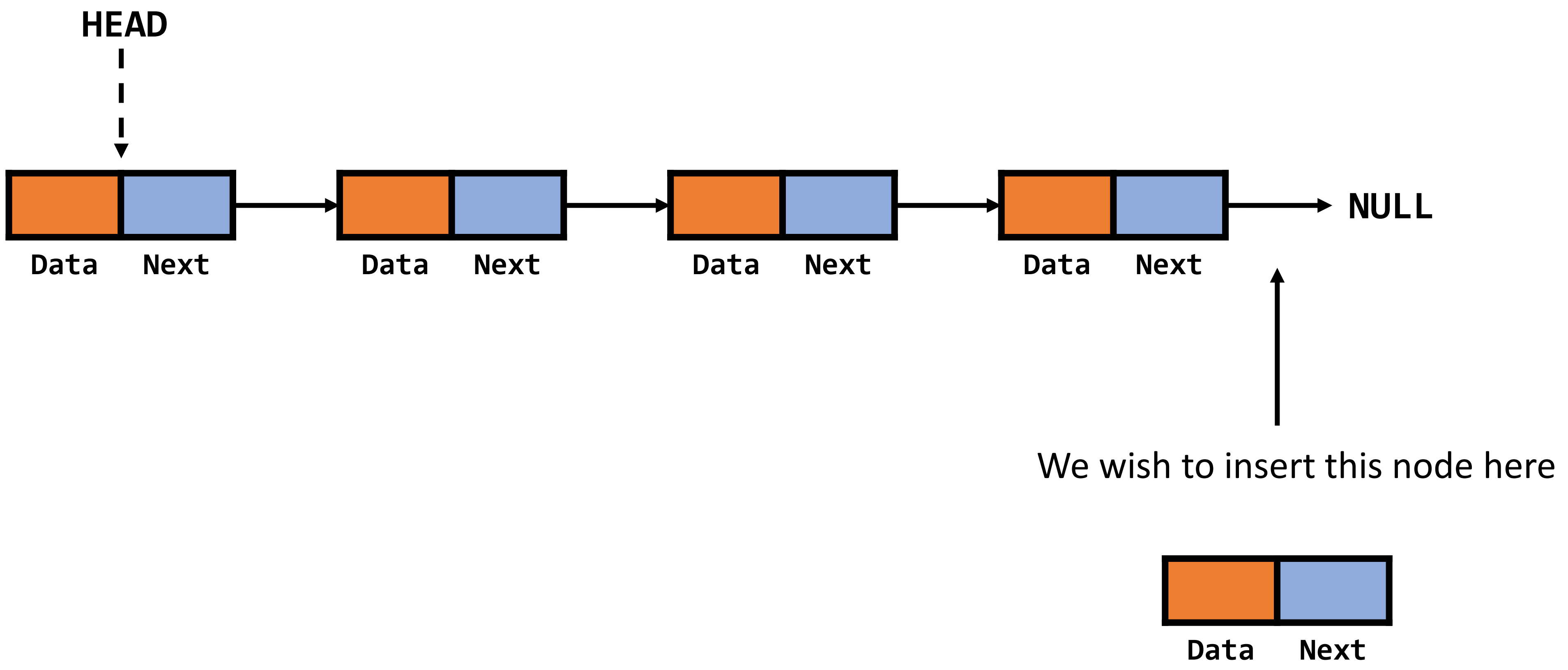
```
# Linked list implementation in Python

class Node:
    """ Represents a single node"""
    def __init__(self, item):
        self.item = item
        self.next = None

class LinkedList:
    """ The whole linked list"""
    def __init__(self):
        self.head = None
        self._size = 0

    def insert_first(self, x):                  # O(1)
        new_node = Node(x)                      # create a new node with item x
        new_node.next = self.head               # set the new node to point to the current head
        self.head = new_node                    # replace the linked list head to be new node
        self._size += 1                         # increase the size

if __name__ == '__main__':

    linked_list = LinkedList()

    # Assign item values
    linked_list.insert_first(1)
    linked_list.insert_first(2)
    linked_list.insert_first(3)

    # list will now contain 3 -> 2 -> 1
```

You can find a full implementation in the module folder that supports all operations.

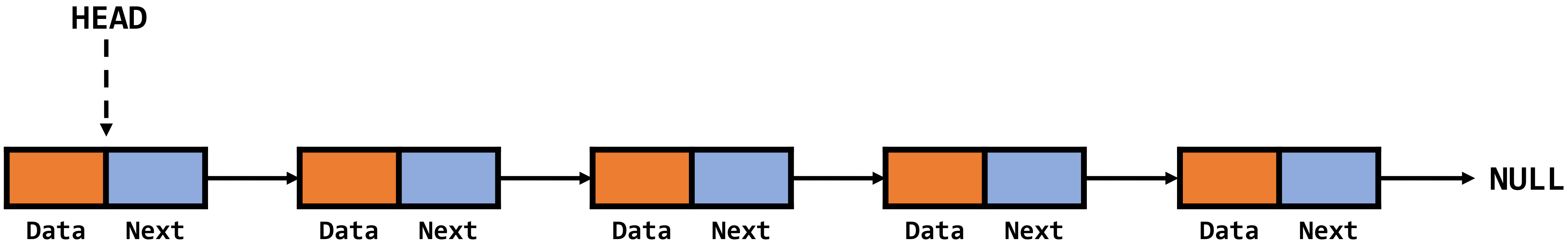
`insert_last()`

Insert at the end of a singly linked list

insert_last()

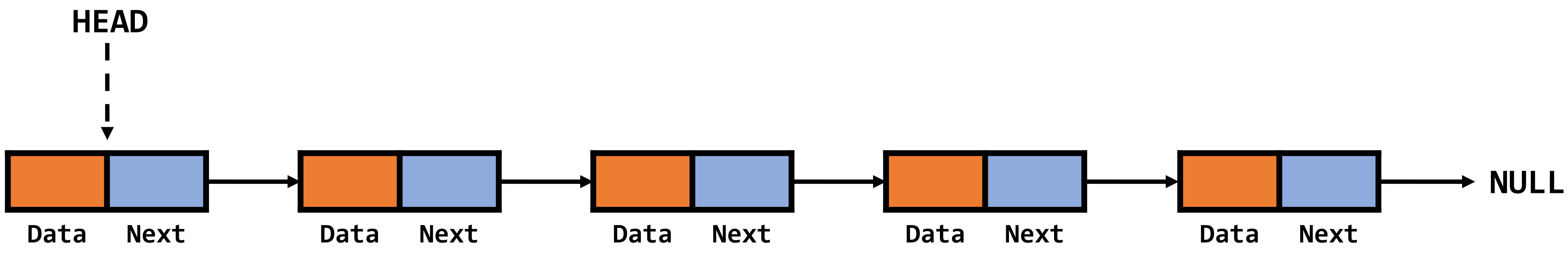

HEAD
Data
Next
Data
Next
Data
Next
Data
Next
NULL
We wish to insert this node here
Data
Next

`insert_last()`



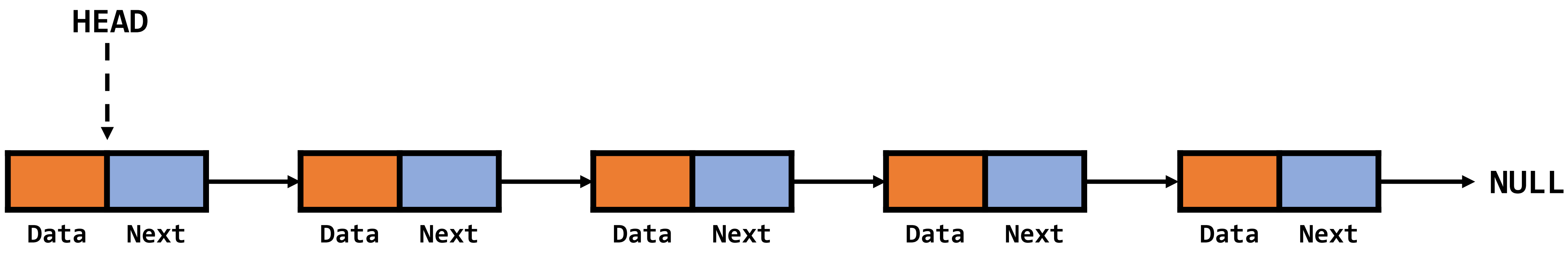
This costs $O(n)$. Why?

`insert_last()`



This costs $O(n)$. Why?

`insert_last()`



This costs $O(n)$. Why?

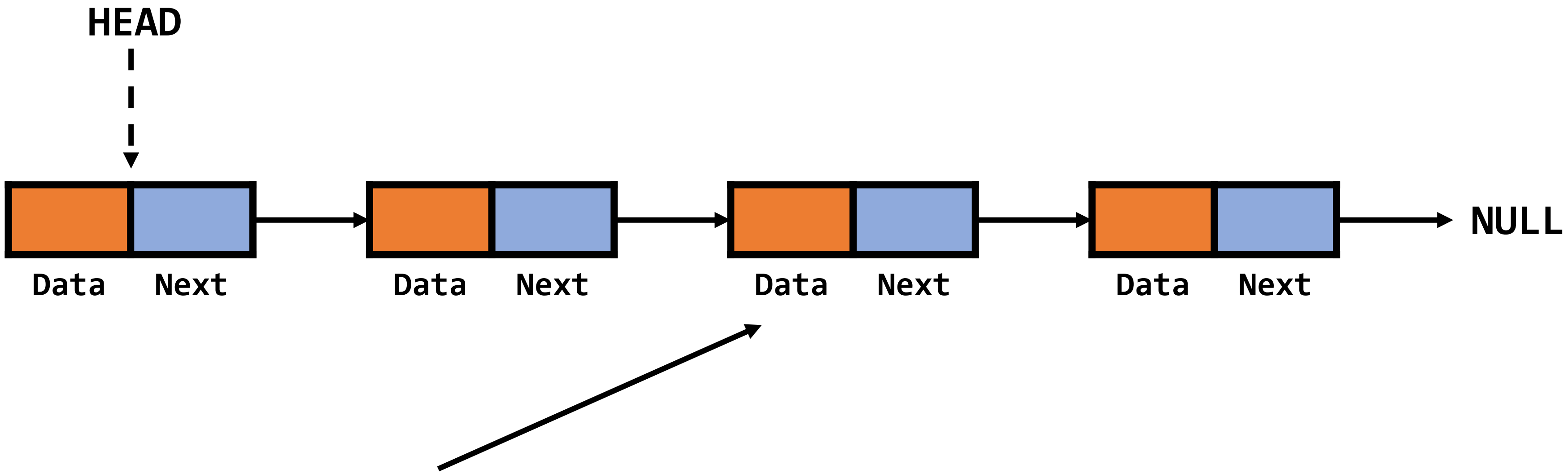
- We traverse the nodes to find the last node
- We set the previous last nodes **next** to the new node
- We set the new nodes **next** to **NULL**

Whilst the insert costs $O(1)$, you have to traverse all the nodes to get to the last node to do the insert. That's $O(n)$.

`get(i)`

Get the node at position i of a singly linked list

get(i)



We wish to **get** the data for this node at position i

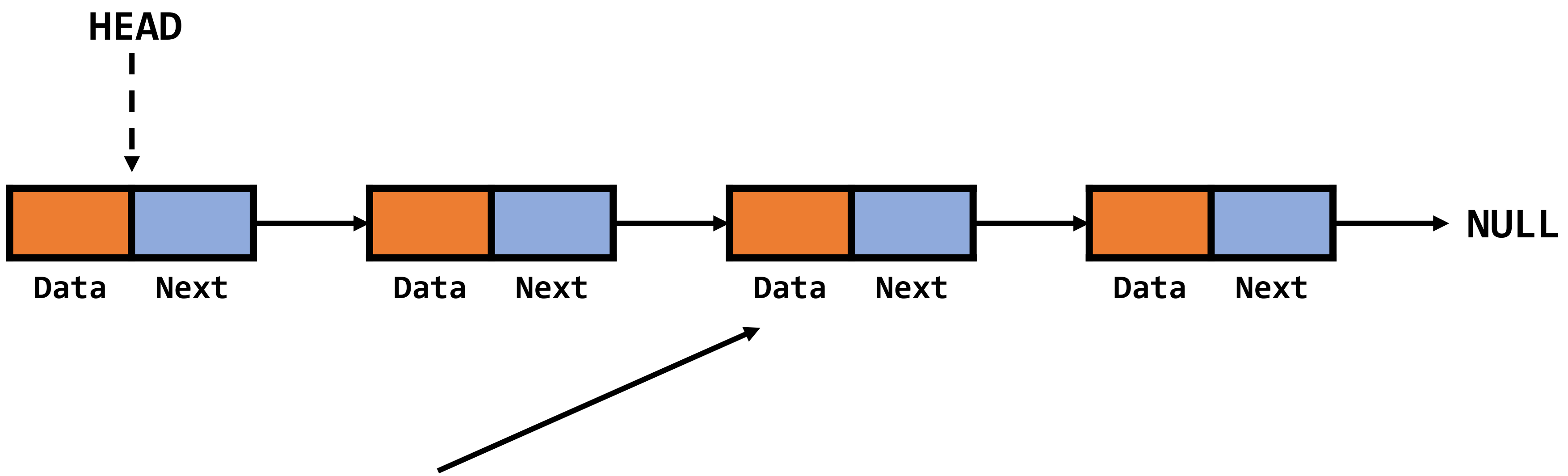
This is $O(i)$ as we have to traverse all items up to this node to access it.

Therefore the worse-case complexity is $O(n)$.

```
set(i,x)
```

Set the node at position i of a singly linked list

`set(i,x)`



We wish to **set** the data for this node at position i

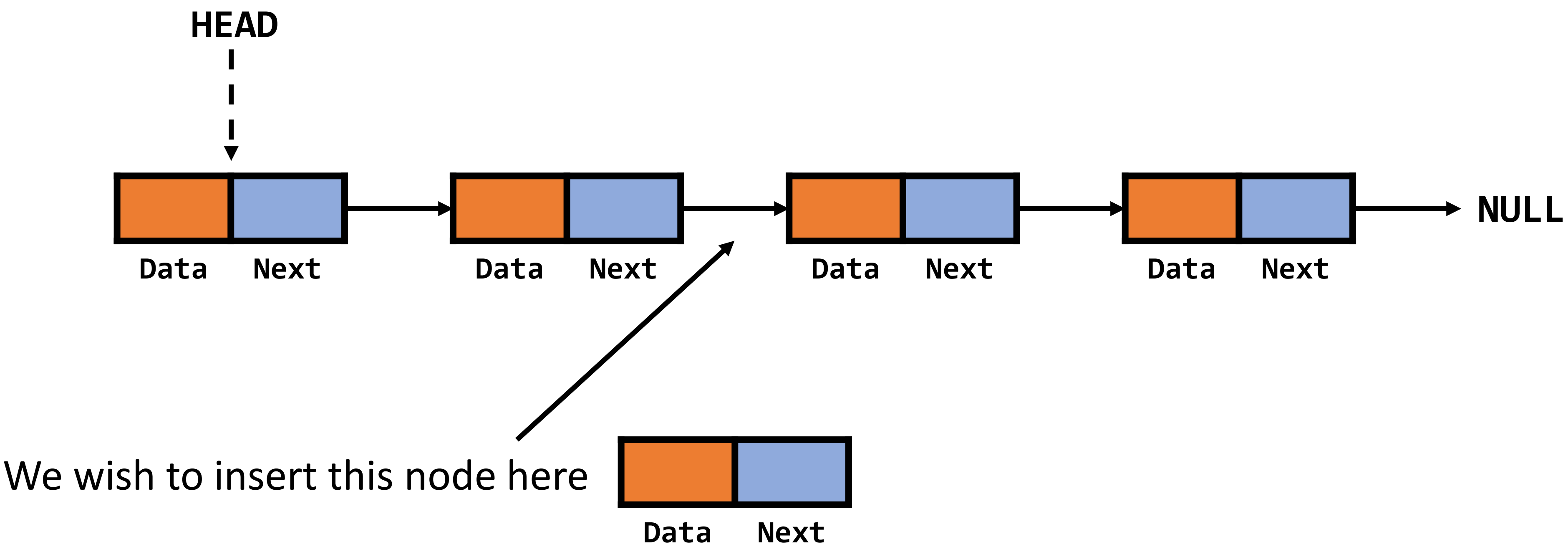
This is $O(i)$ as we have to traverse all items up to this node to then set the data.

Therefore the worse-case complexity is $O(n)$.

`insert(i,x)`

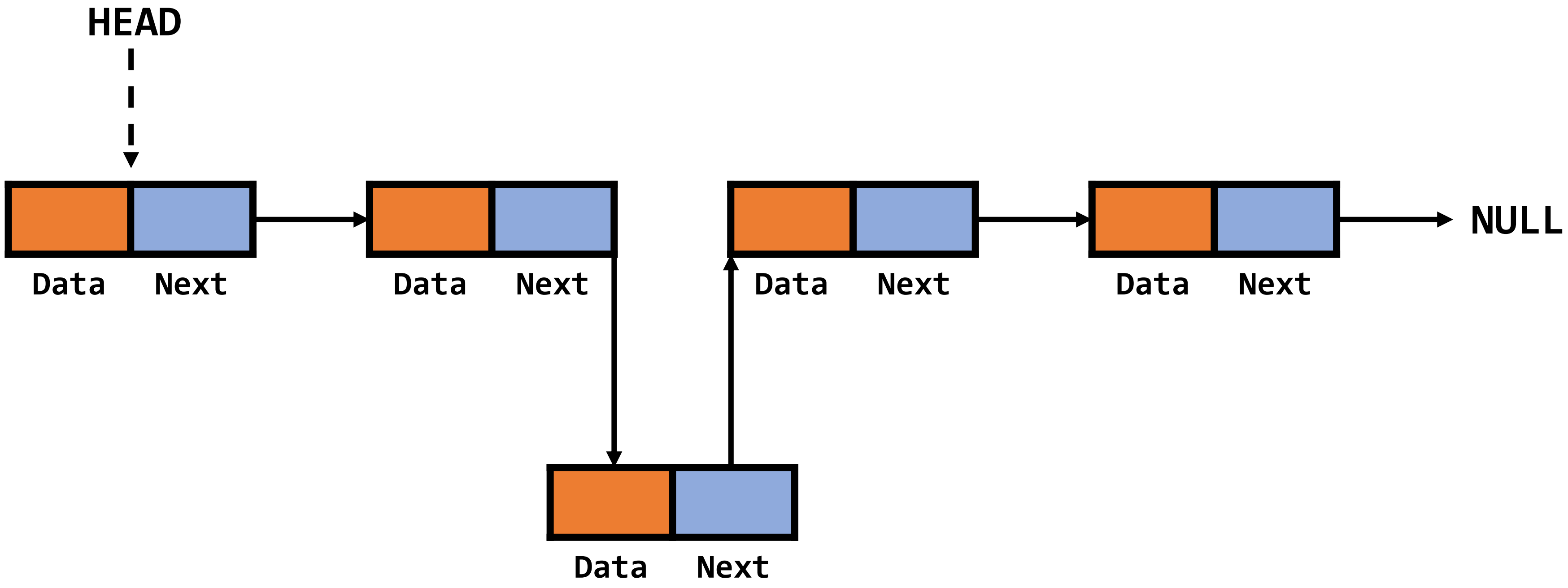
Insert at position i of a singly linked list

insert(i,x)



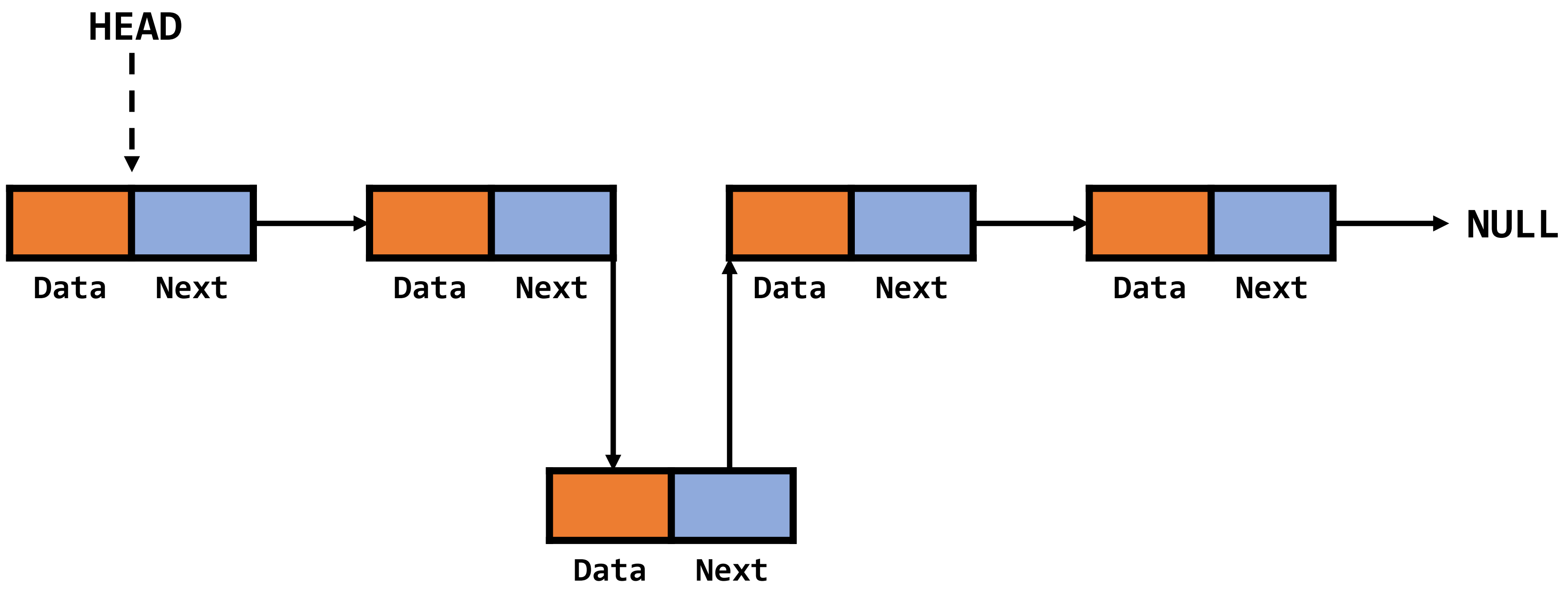
HEAD
Data
Next
Data
Next
Data
Next
Data
Next
NULL
We wish to insert this node here
Data
Next

insert(i,x)



This costs $O(i)$. Where i is the position we insert at. Why?

`insert(i,x)`



This costs $O(i)$. Where i is the position we insert at. Why?

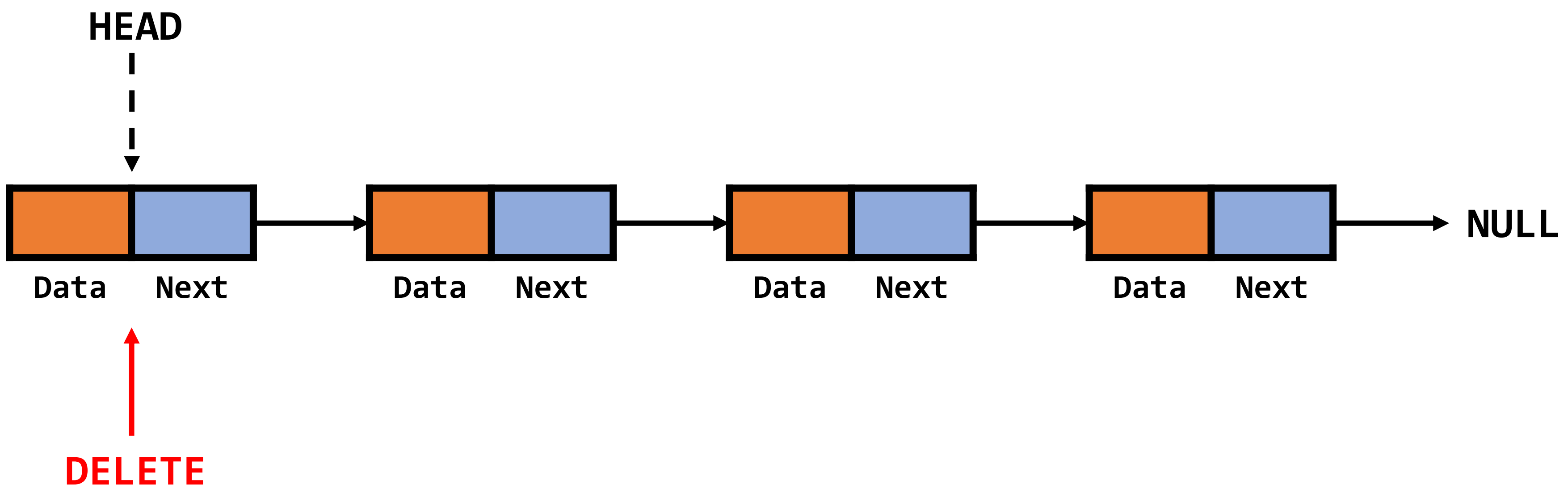
Whilst the insert costs $O(1)$, you have to traverse all the nodes to get to i. That's $O(i)$

Thus in general the worse-case time complexity for `insert()` is $O(n)$

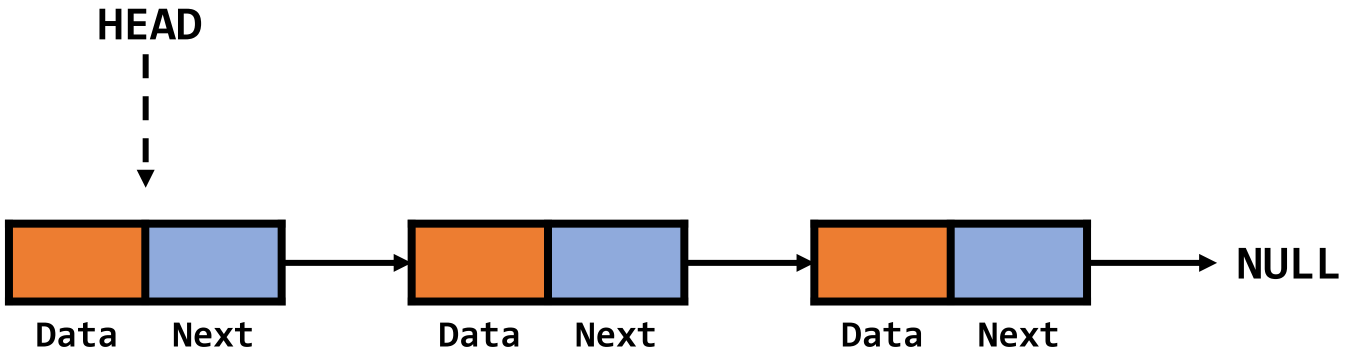
`delete_first()`

Delete at the beginning of a singly linked list

delete_first()

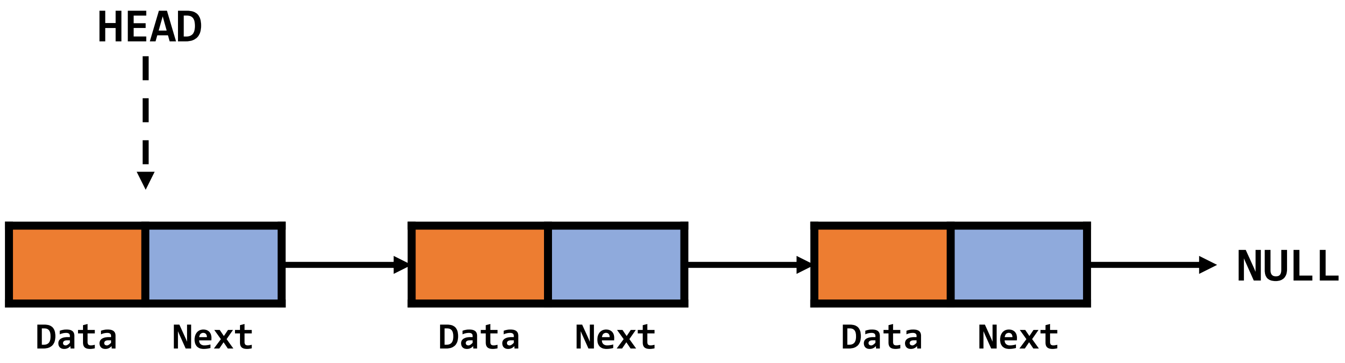


`delete_first()`



This costs $O(1)$. Why?

`delete_first()`



This costs $O(1)$. Why?

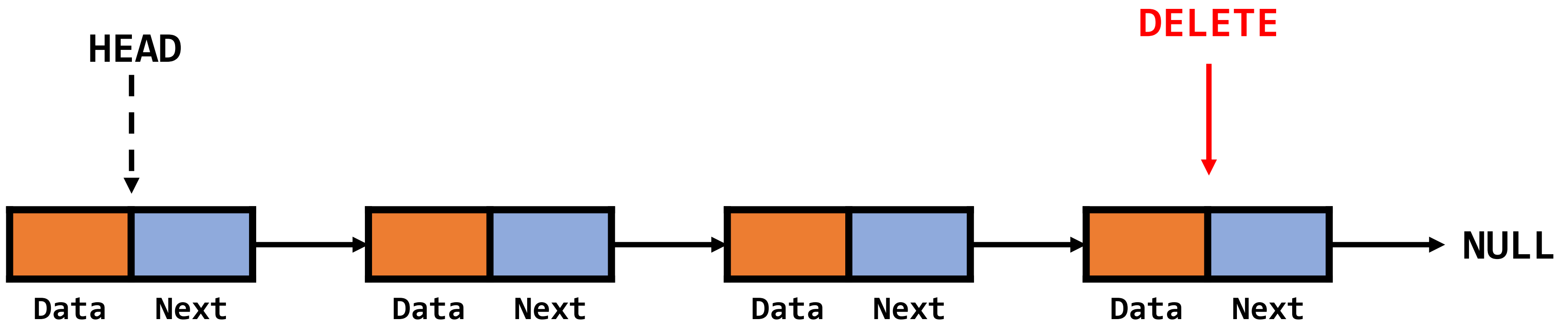
We are storing the **HEAD** of the linked list, which means we know what the first node is so we can just delete it.

We reassign the second node as the **HEAD** and remove the first node from memory.

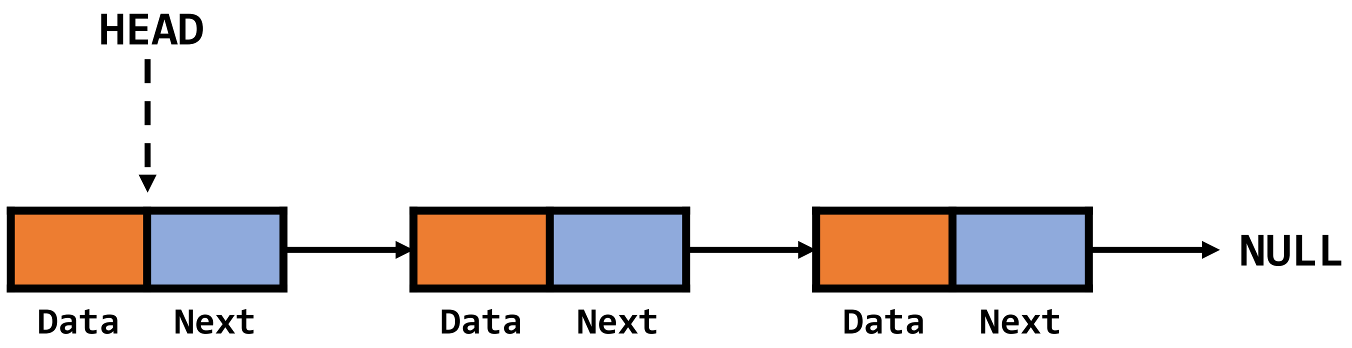
`delete_last()`

Delete at the end of a singly linked list

delete_last()

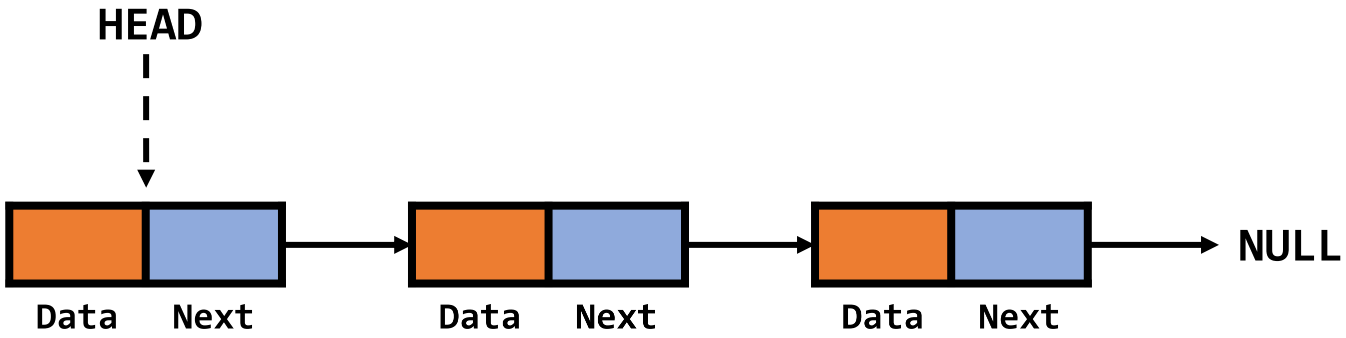


`delete_last()`



This costs $O(n)$. Why?

`delete_last()`



This costs $O(n)$. Why?

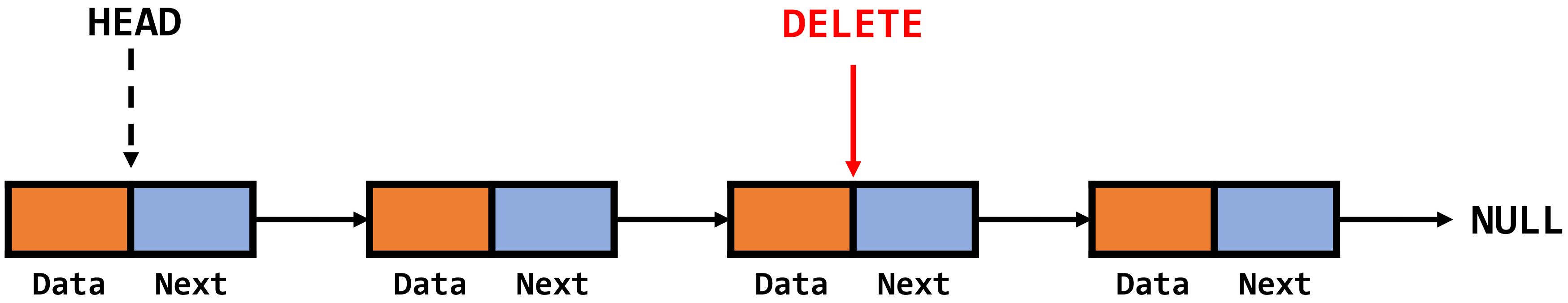
- We traverse the nodes to find the second to last node.
- Then we remove the last node from memory
- We set the second to last nodes **next** to **NULL**

Whilst the delete costs $O(1)$, you have to traverse all the nodes to get to the last node to do the delete. That's $O(n)$.

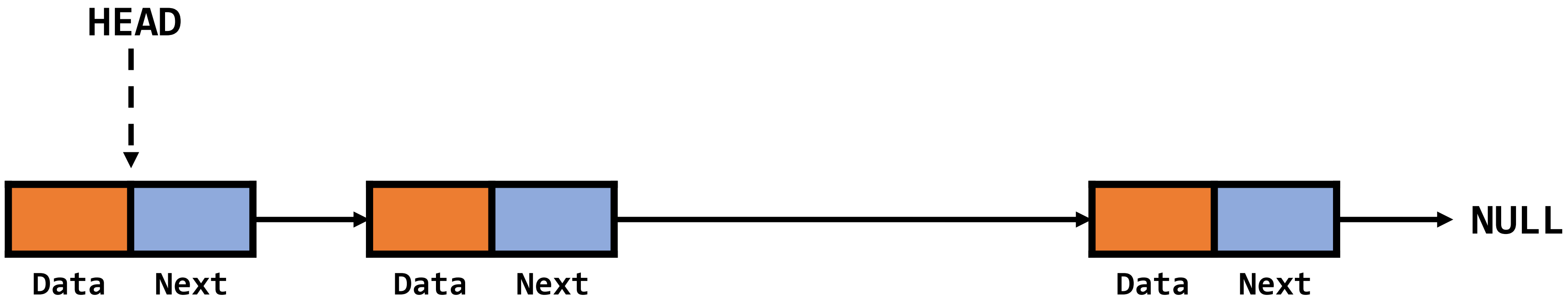
`delete(i)`

Delete at position i of a singly linked list

delete(i)



delete(i)



This costs $O(i)$. Where i is the position we delete at. Why?

`delete(i)`

This costs $O(i)$. Where i is the position we delete at. Why?

Whilst the insert costs $O(1)$, you have to traverse all the nodes to get to i. That's $O(i)$

Thus in general the worse-case time complexity for `delete()` is $O(n)$

Singly Linked List Summary

Data Structure	`create(X)`	`get(i)` `set(i,x)`	`insert(i,x)` `delete(i)`	`insert_first(i,x)` `delete_first()`	`insert_last(i,x)` `delete_last()`	Space
Singly Linked List	$O(n)$	$O(n)$	$O(n)^{\dagger}$	$O(1)$	$O(n)$	$O(n)$

Worse-case Complexity

† assumes traversal to ith node

Shuffling Cards

How many ways are there to shuffle a deck of cards?

Shuffling Cards

How many ways are there to shuffle a deck of cards?

$$52! = 52 \times 51 \times 50 \times \cdots \times 1 \approx 8.065 \times 10^{67}$$

Approximately…

8000000000000000000000000000000000000000000000000
0000000000000000000

Much larger than the estimated number of atoms on the planet!

Doubly Linked List

Motivation

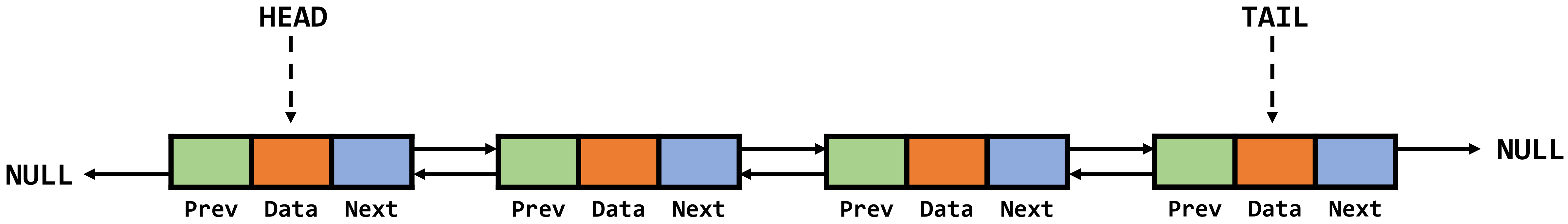
- We want to insert and delete from the back
- We want to improve our access time (`get()/set()`)
- We want to traverse backwards and forwards

Solution

Store a link to the **previous** node as well as the **next** node. Also store last node (**TAIL**).

Note we can store the **TAIL** in a singly linked list which will make insertion and deletion at the end of the list $O(1)$. Why?

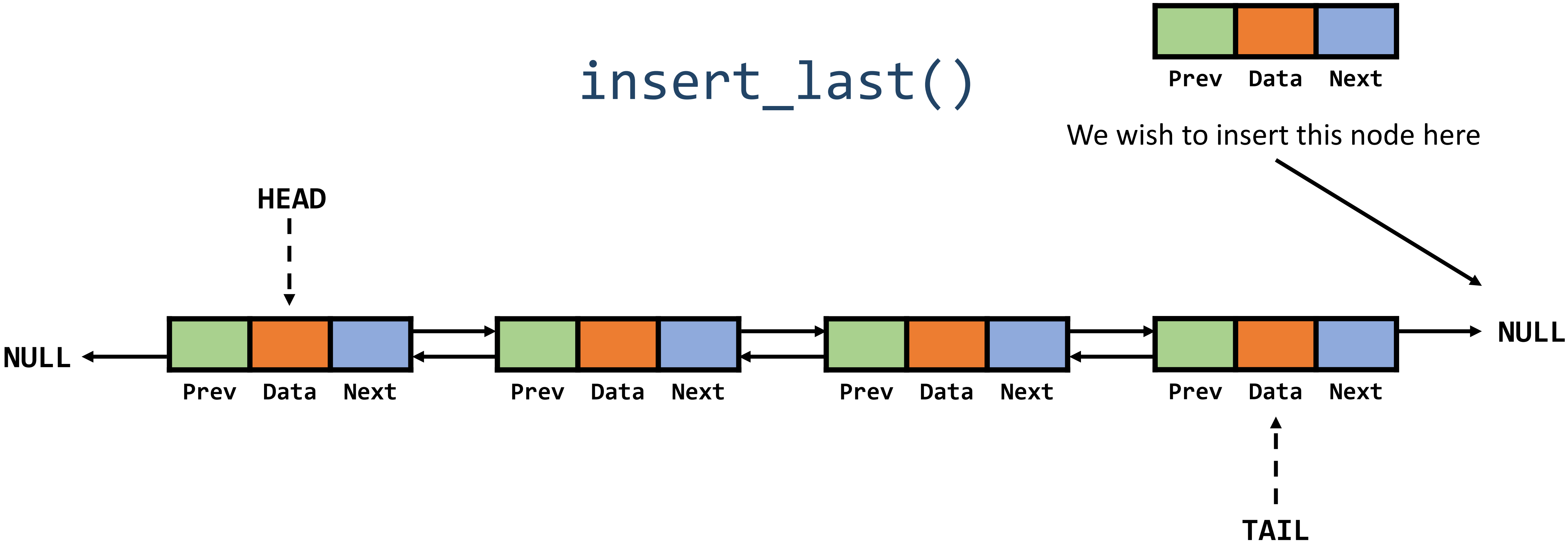
Doubly Linked List



`insert_last()`

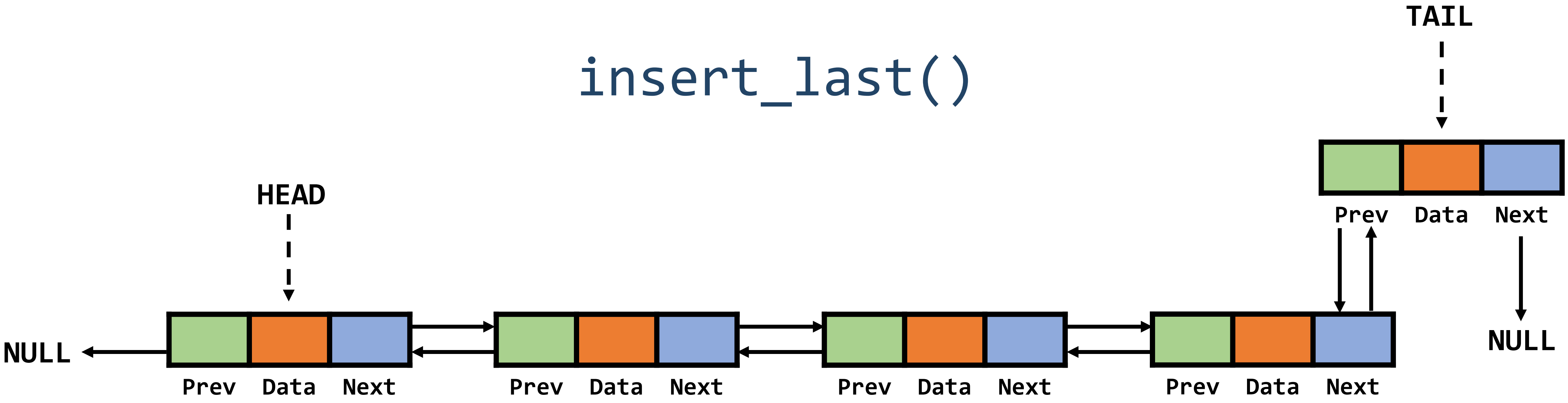
Insert at the end of a doubly linked list

insert_last()



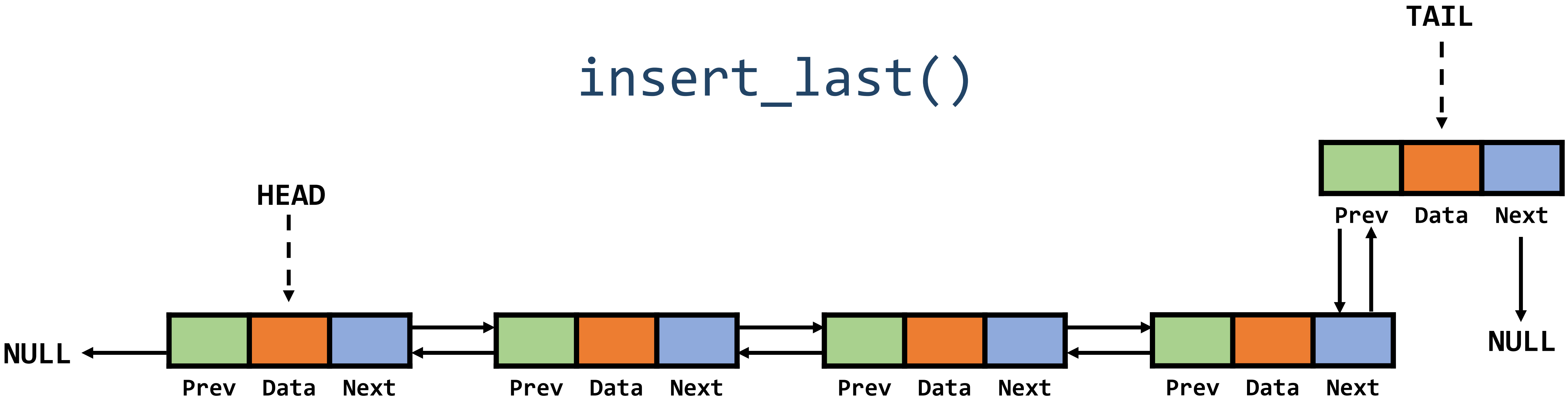
Sensitivity: Internal

insert_last()



This costs $O(1)$. Why?

insert_last()



This costs $O(1)$. Why?

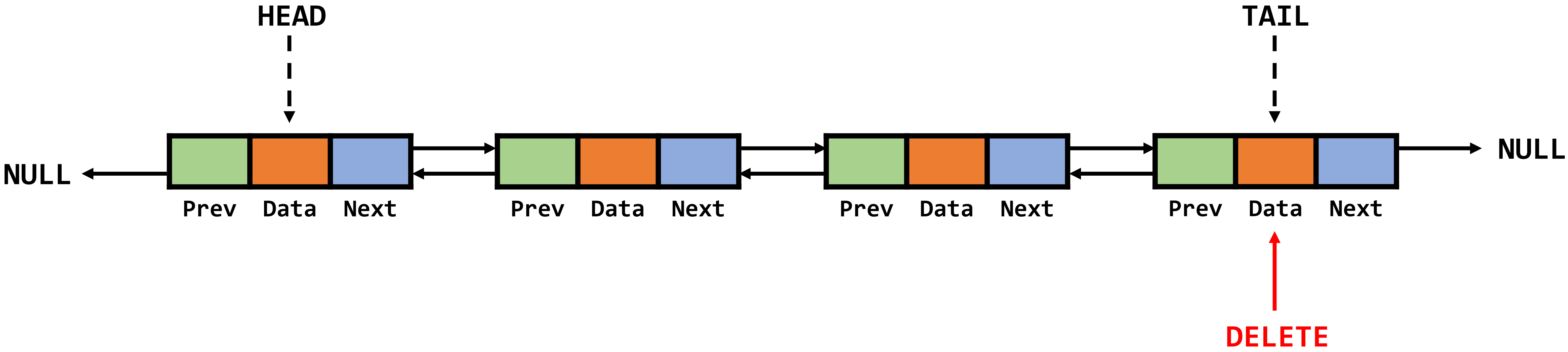
We are now storing the **TAIL** of the linked list, which means we know what the last node is so we can just insert it.

We reassign the new node as the **TAIL** and set its **previous** to be the previous **TAIL** and its **next** to **NULL**

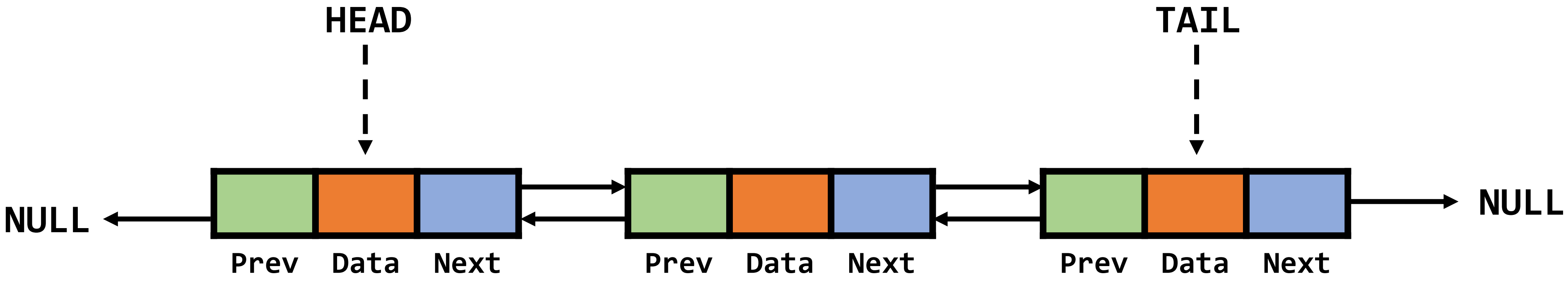
`delete_last()`

Delete at the end of a doubly linked list

delete_last()

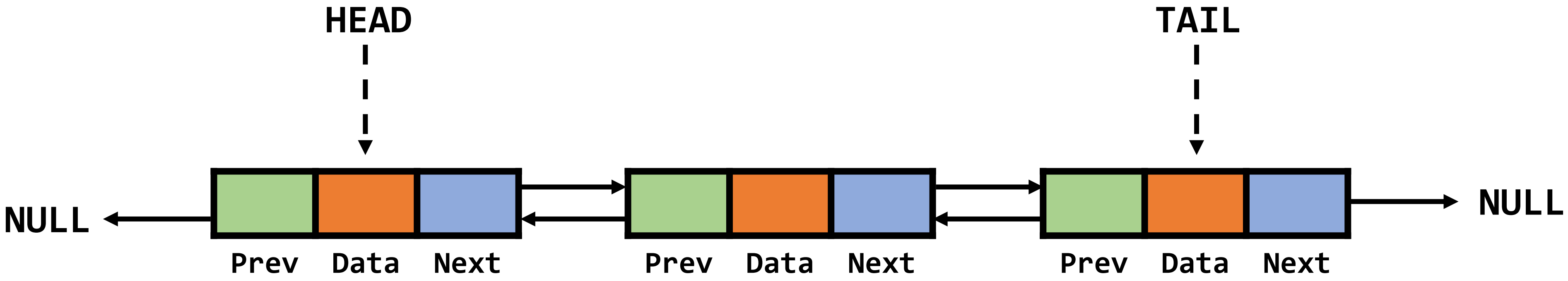

HEAD
TAIL
NULL
Prev
Data
Next
Prev
Data
Next
Prev
Data
Next
Prev
Data
Next
NULL
DELETE

delete_last()



This costs $O(1)$. Why?

delete_last()



This costs $O(1)$. Why?

We are now storing the **TAIL** of the linked list, which means we know what the last node is so we can just delete it.

We reassign the new node as the **TAIL** and set its **next** to **NULL**

Doubly Linked List Summary

- Insertion and deletion at front and back is $O(1)$
- Can traverse both ways

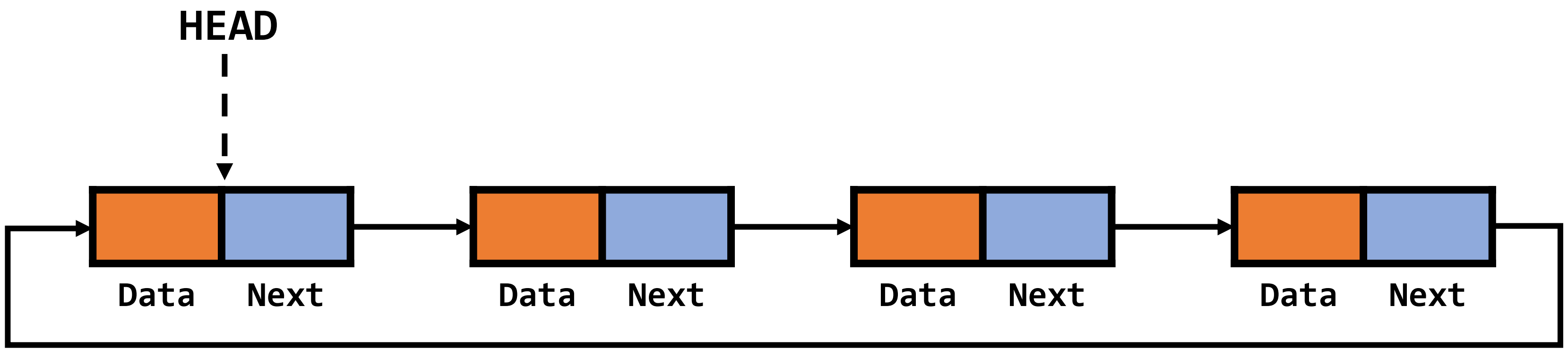
Data Structure	`create(X)`	`get(i)` `set(i,x)`	`insert(i,x)` `delete(i)`	`insert_first(i,x)` `delete_first()`	`insert_last(i,x)` `delete_last()`	Space
Doubly Linked List	$O(n)$	$O(n)$	$O(n)^\dagger$	$O(1)$	$O(1)$	$O(n)$

Worse-case Complexity

† assumes traversal to ith node

Circular Linked List

Circular Linked List



We won't look at this in detail.

- **TAIL** node points back to **HEAD**
- Therefore any node can be the HEAD

There is also a doubly circular linked list.

Try pressing ALT-TAB (Windows/Linux) or COMMAND-TAB (Mac)


Singly Linked List
HEAD
Data
Next
Data
Next
Data
Next
Data
Next
NULL
Doubly Linked List
HEAD
TAIL
NULL
Prev
Data
Next
Prev
Data
Next
Prev
Data
Next
Prev
Data
Next
NULL
Circular Linked List
HEAD
Data
Next
Data
Next
Data
Next
Data
Next

Lecture Summary

- Sequence
- Linear data structure
- Can support all sequence operations
- Not indexed
- Good for inserts and deletions, not for get and set (access).
- Good for dynamic allocation of space

Lecture Summary

Data Structure	`create(X)`	`get(i)` `set(i,x)`	`insert(i,x)` `delete(i)`	`insert_first(i,x)` `delete_first()`	`insert_last(i,x)` `delete_last()`	Space
Array	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Dynamic Array	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)^{**}$	$O(n)$
Singly Linked List	$O(n)$	$O(n)$	$O(n)^{\dagger}$	$O(1)$	$O(n)$	$O(n)$
Doubly Linked List	$O(n)$	$O(n)$	$O(n)^{\dagger}$	$O(1)$	$O(1)$	$O(n)$

Worse-case Complexity

**Amoritized Time, † assumes traversal to ith node