Topic 12 – Data Structures Part 3 : Linked List| Stacks|Queues

tafeSA

Government of South Australia

Recap – The Python List (How data is stored)

```
my_list = [1,1,2,3]
```

Array Memory
(contiguous)

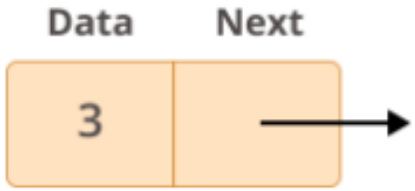
Other Random Memory
(not contiguous)

List Index	Address	Data
0	0x00	
1	0x01	
2	0x02	
3	0x03	

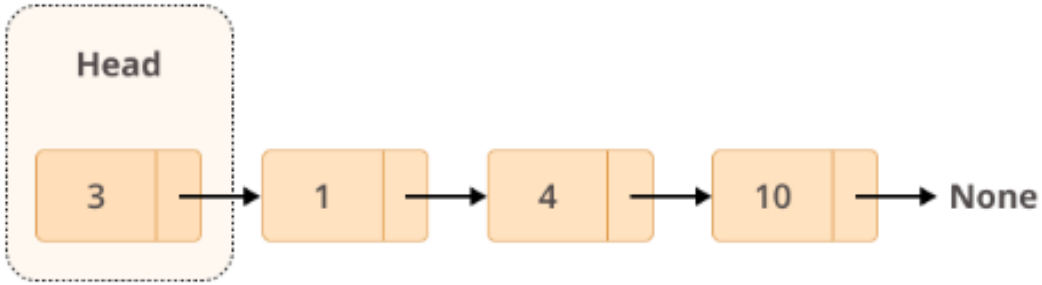
Address	Data
0x50	1
0x22	1
0x9E	2
0xF2	3

Linked List – Main Concepts

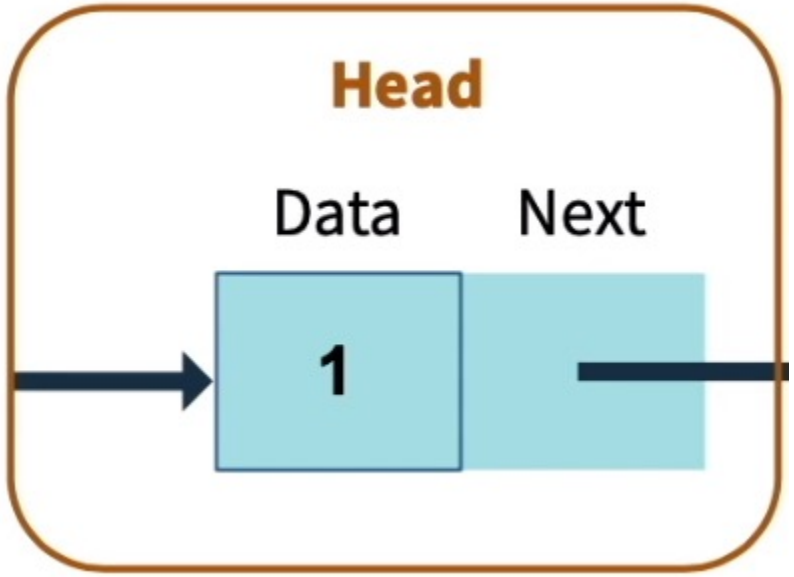
- Each element in a list is called a node
- Every node has two different fields:
  1. Data contains the value to be stored in the node
  2. Next contains a reference to the next node on the list



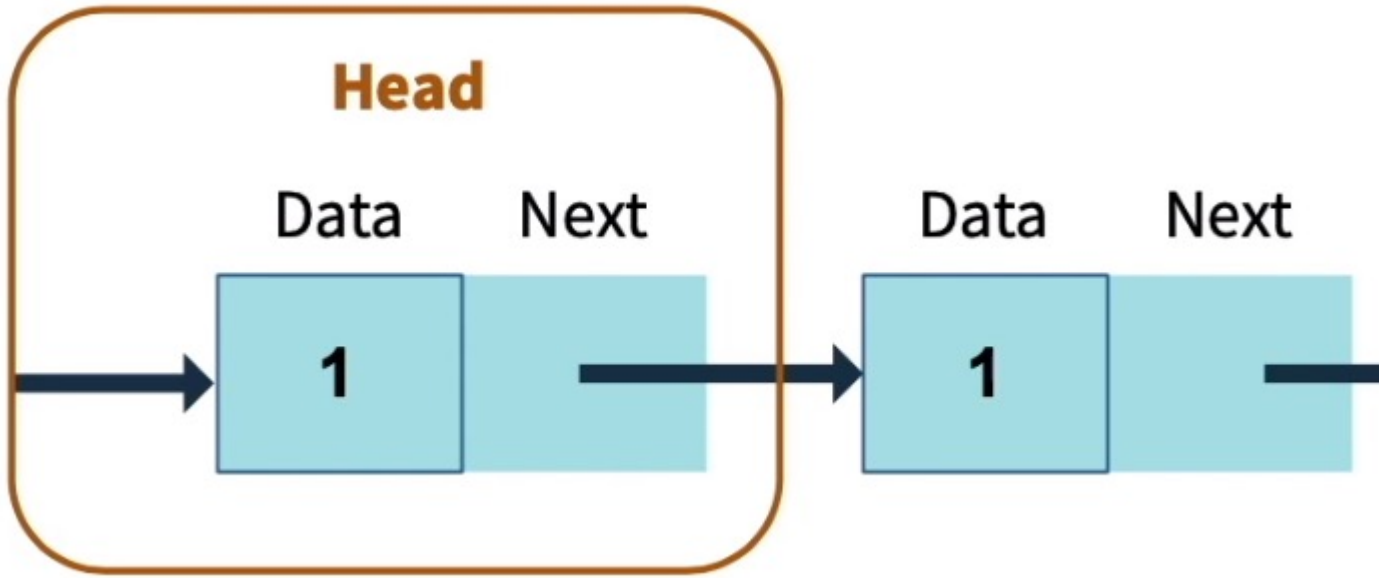
- A Linked List is a collection of nodes, the first node is called the head, and used as starting point when iterating through the list
- The last node.next reference points to None



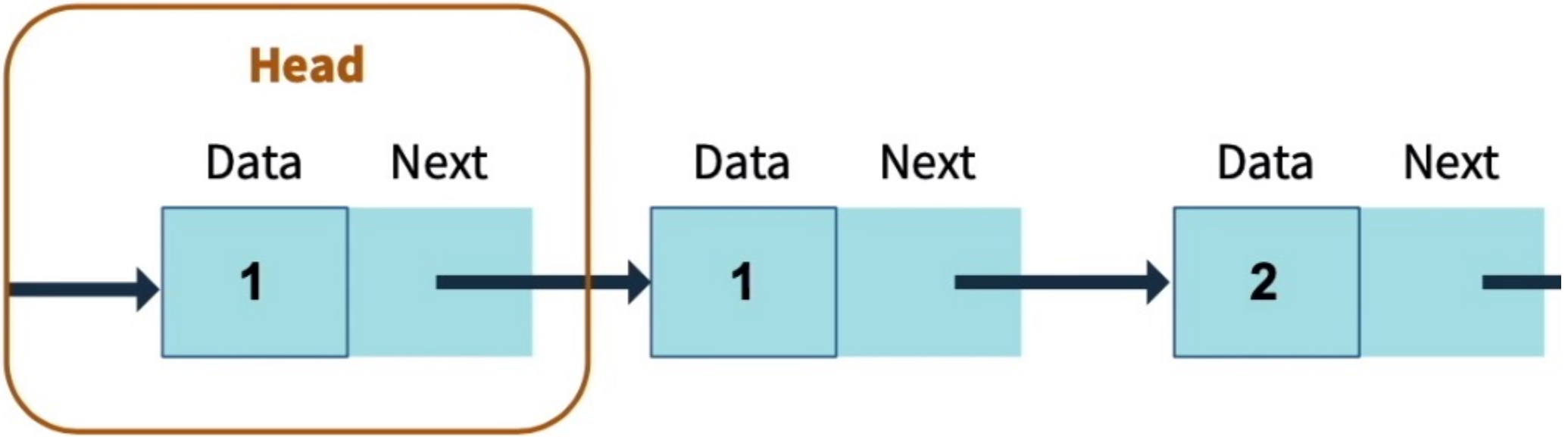
The Linked List (Start at Head)

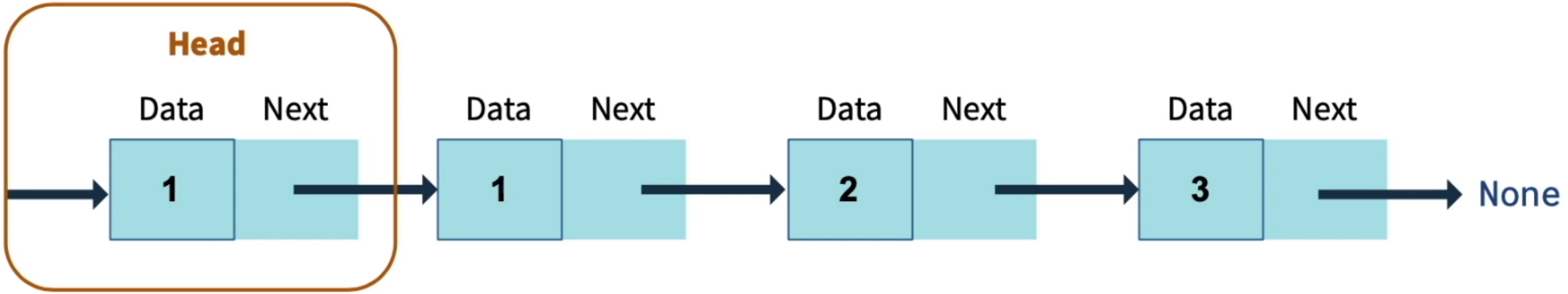


Next Element



Accessing Elements in the Linked List




Head
Data
Next
1
Data
Next
1
Data
Next
2
Data
Next
3
None

Performance Comparisons
List vs Linked Lists (LL)

- Inserting / Removing elements at beginning and end of list using append() and pop() – is a constant time: O(1) operation

  (Contrast this with its insert() and remove() methods – O(n))

- Linked List insertion and deletion (if there is a tail node) of elements is also a constant time O(1) operation

- Linked List implemented as a Queue have a performance advantage over normal lists when removing and adding elements to the structure.

- Lists perform better (O(1))than LL when looking up an element you want to access (O(n))

- When searching for a specific element, both lists and LL perform similarly, with a time complexity of O(n)

Comparing List with Linked List

List are better at:

- finding elements in the list based on the index
- Appending items at the end of the list

List are inefiicient at:

- Appending elements in missdle or worse case at the beginning

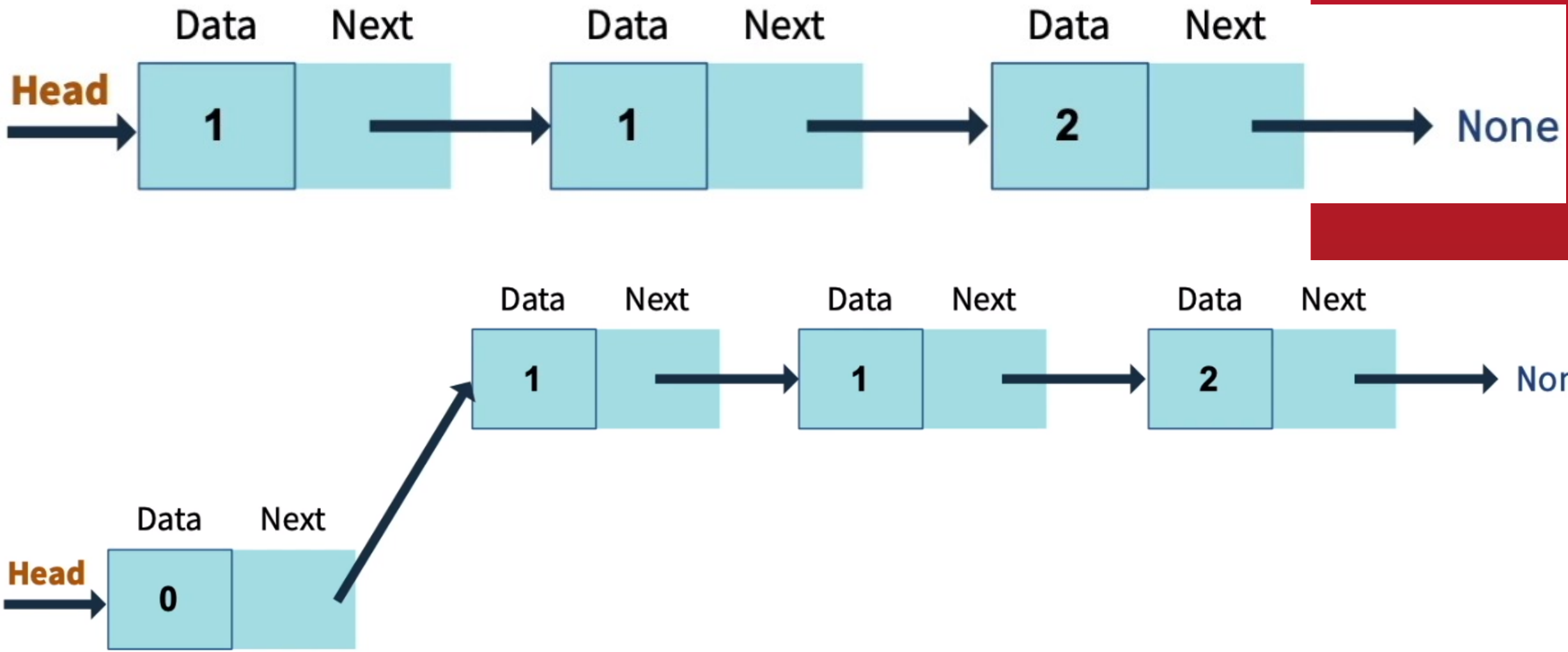
Linked List are better at:

- Inserting items at the beginning of the list or end of list

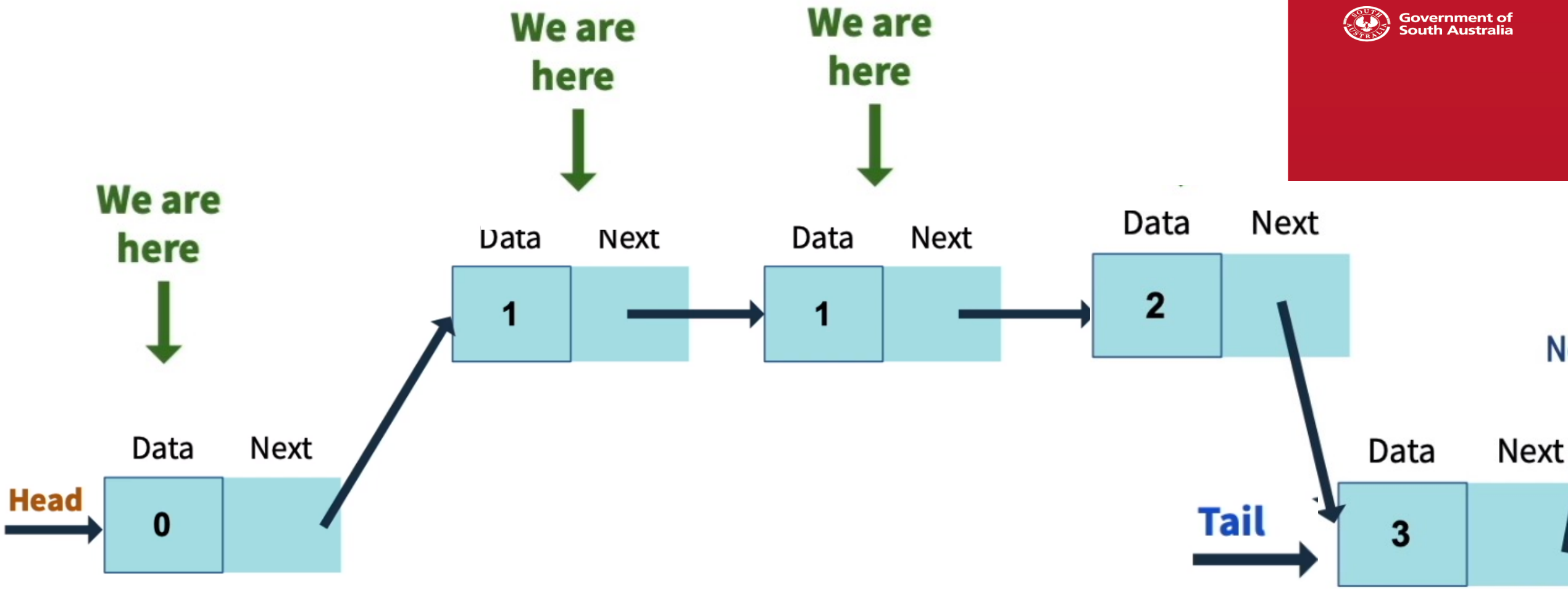
The Linked List is NOT an Array

- Linked lists are not represented by arrays in memory.
- Nodes are stored in random memory locations.
  - Nodes are represented as objects, often created from a class or dataclass.
- Because no arrays are involved, linked lists never need to be resized!

Inserting a Node at the Head of the Liked List



Inserting a Node at the end of the Linked List



Lists vs Linked List – Recap..

List	Linked List
Fast for accessing an element at a specific index.	**Slower** at accessing an element at a specific index.
Slower at inserting elements in the middle or beginning.	**Fast** for inserting or accessing elements at the beginning or end.

Creating a Linked List

- Create a class to represent your linked list (LL)

```
class LinkedList:
    def __init__(self):
        self.head = None
```

- Next, create another class to represent the head node of the linked list (The head is the only info the LL needs to know)

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Putting it together

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    def __repr__(self):
        return self.data

class LinkedList:
    def __init__(self):
        self.head = None

    def __repr__(self):
        node = self.head
        nodes = []
        while node is not None:
            nodes.append(node.data)
            node = node.next
        nodes.append("None")
        return " -> ".join(nodes)
```

Can also use a __str__()

```
>>> llist = LinkedList()
>>> llist
None

>>> first_node = Node("a")
>>> llist.head = first_node
>>> llist
a -> None

>>> second_node = Node("b")
>>> third_node = Node("c")
>>> first_node.next = second_node
>>> second_node.next = third_node
>>> llist
a -> b -> c -> None
```

tafeSA Government of South Australia

Creating a Linked List with data from a List

Make a slight change to the LL __init__() that allows the quick creating of LL nodes with data from a List

```
llist = LinkedList(["a", "b", "c", "d", "e"])
```

```
def __init__(self, nodes=None):
    self.head = None
    if nodes is not None:
        node = Node(data=nodes.pop(0))
        self.head = node
        for elem in nodes:
            node.next = Node(data=elem)
            node = node.next
```

Traversing a Linked List

- Traversing means going through every single node starting with the **head** and ending with the node -> next Value is **None**

- To iterate through a LL like a List define the __iter__() (see below)

```
def __iter__(self):
    node = self.head
    while node is not None:
        yield node
        node = node.next
```

```
llist = LinkedList(["a", "b", "c", "d", "e"])
```

```
for node in llist:
    print(node)
```

```
a
b
c
d
e
```

Inserting a New Node at the Head of a Linked List

Create a new new node and point the head of the list to it

```
def add_first(self, node):
    node.next = self.head
    self.head = node
```

```
>>> llist = LinkedList()
>>> llist
None

>>> llist.add_first(Node("b"))
>>> llist
b -> None

>>> llist.add_first(Node("a"))
>>> llist
a -> b -> None
```

tafeSA
Government of South Australia

Inserting at the End of a Linked List

You have to traverse the entire Linked List to find the last element (node.next = None)

```
def add_last(self, node):
    if self.head is None:
        self.head = node
        return
    for current_node in self:
        pass
    current_node.next = node
```

This loop will reach the end of the list (last node)

The current_node is now pointing to the last node on the list

```
>>> llist = LinkedList(["a", "b", "c", "d"])
>>> llist
a -> b -> c -> d -> None

>>> llist.add_last(Node("e"))
>>> llist
a -> b -> c -> d -> e -> None

>>> llist.add_last(Node("f"))
>>> llist
a -> b -> c -> d -> e -> f -> None
```

Inserting a Node AFTER an existing Node

```
def add_after(self, target_node_data, new_node):
    if self.head is None:
        raise Exception("List is empty")

    for node in self:
        if node.data == target_node_data:
            new_node.next = node.next
            node.next = new_node
            return

    raise Exception("Node with data '%s' not found" % target_node_data)
```

```
>>> llist = LinkedList()
>>> llist.add_after("a", Node("b"))
Exception: List is empty

>>> llist = LinkedList(["a", "b", "c", "d"])
>>> llist
a -> b -> c -> d -> None

>>> llist.add_after("c", Node("cc"))
>>> llist
a -> b -> c -> cc -> d -> None

>>> llist.add_after("f", Node("g"))
Exception: Node with data 'f' not found
```

Inserting a Node BEFORE an existing Node

```
def add_before(self, target_node_data, new_node):
    if self.head is None:
        raise Exception("List is empty")

    if self.head.data == target_node_data:
        return self.add_first(new_node)

    prev_node = self.head
    for node in self:
        if node.data == target_node_data:
            prev_node.next = new_node
            new_node.next = node
            return
        prev_node = node

    raise Exception("Node with data '%s' not found" % target_node_data)
```

```
>>> llist = LinkedList()
>>> llist.add_before("a", Node("a"))
Exception: List is empty

>>> llist = LinkedList(["b", "c"])
>>> llist
b -> c -> None

>>> llist.add_before("b", Node("a"))
>>> llist
a -> b -> c -> None

>>> llist.add_before("b", Node("aa"))
>>> llist.add_before("c", Node("bb"))
>>> llist
a -> aa -> b -> bb -> c -> None

>>> llist.add_before("n", Node("m"))
Exception: Node with data 'n' not found
```

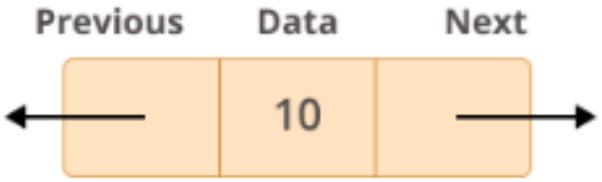
Removing a Node from Linked List

```
def remove_node(self, target_node_data):
    if self.head is None:
        raise Exception("List is empty")

    if self.head.data == target_node_data:
        self.head = self.head.next
        return

    previous_node = self.head
    for node in self:
        if node.data == target_node_data:
            previous_node.next = node.next
            return
        previous_node = node

    raise Exception("Node with data '%s' not found" % target_node_data)
```

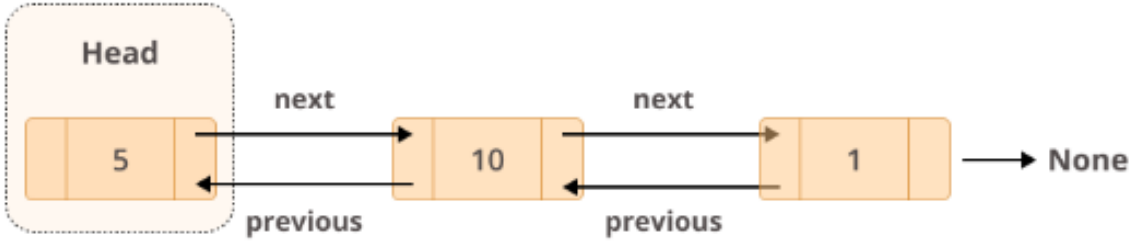
```
>>> llist = LinkedList()
>>> llist.remove_node("a")
Exception: List is empty

>>> llist = LinkedList(["a", "b", "c", "d", "e"])
>>> llist
a -> b -> c -> d -> e -> None

>>> llist.remove_node("a")
>>> llist
b -> c -> d -> e -> None

>>> llist.remove_node("e")
>>> llist
b -> c -> d -> None

>>> llist.remove_node("c")
>>> llist
b -> d -> None

>>> llist.remove_node("a")
Exception: Node with data 'a' not found
```

Using Advanced Linked Lists

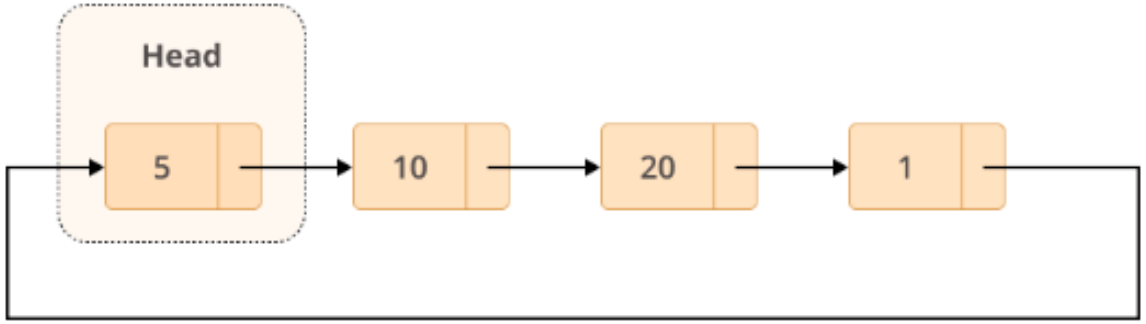
- Doubly Linked Lists




```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.previous = None
```



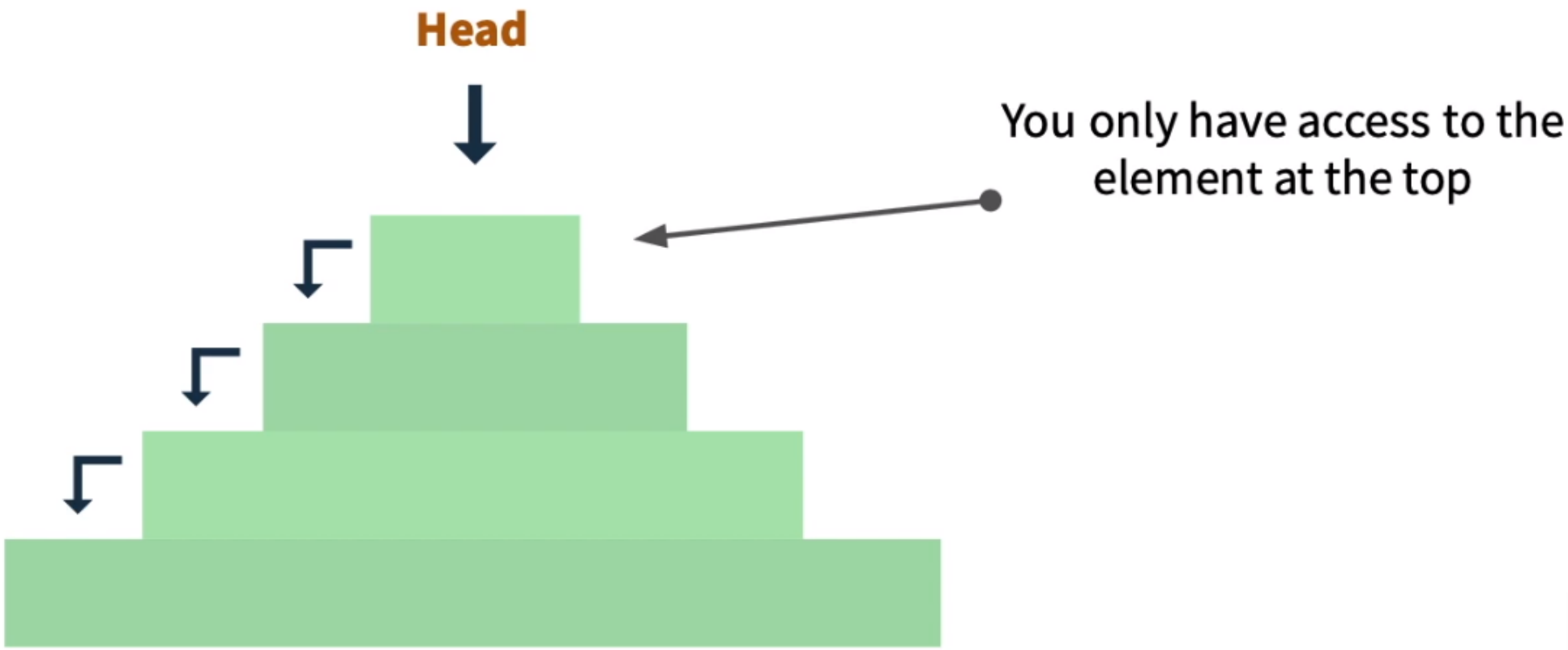
- Circular Linked Lists



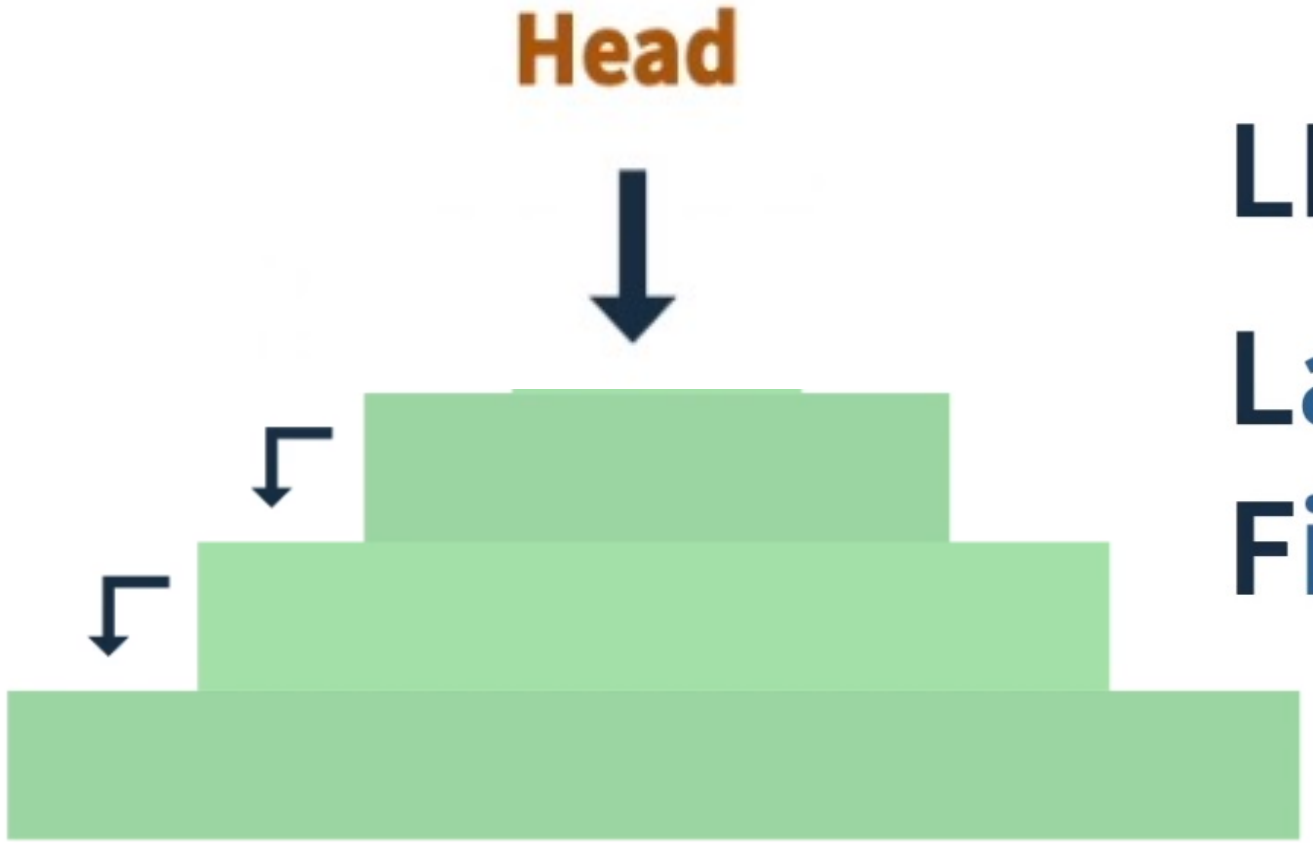
The Stack

tafeSA
Government of South Australia

The Stack



The Stack – Accessing other elements

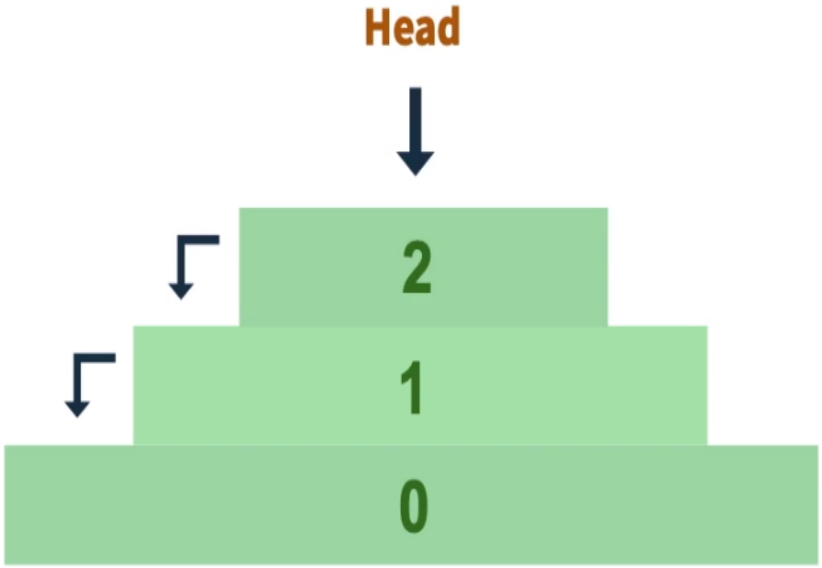


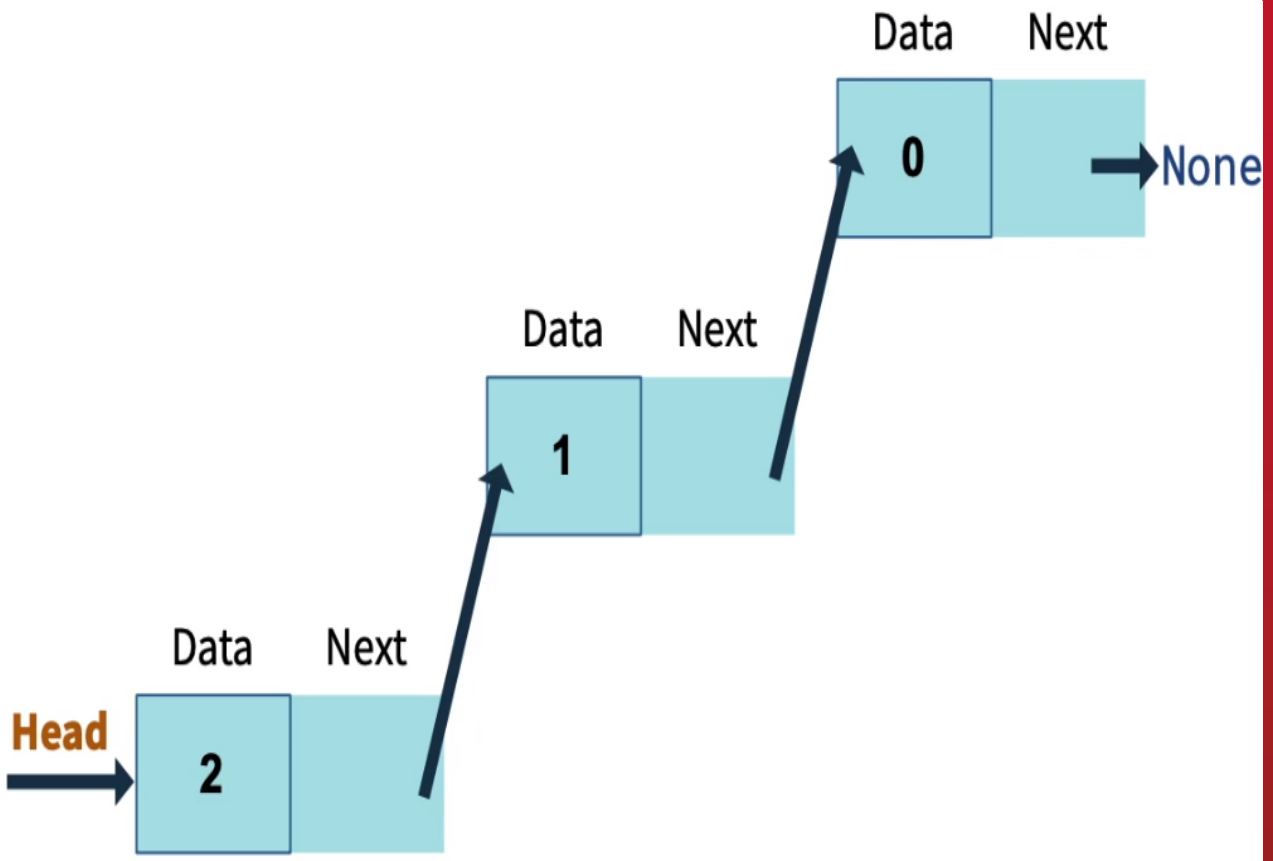
LIFO

Last In

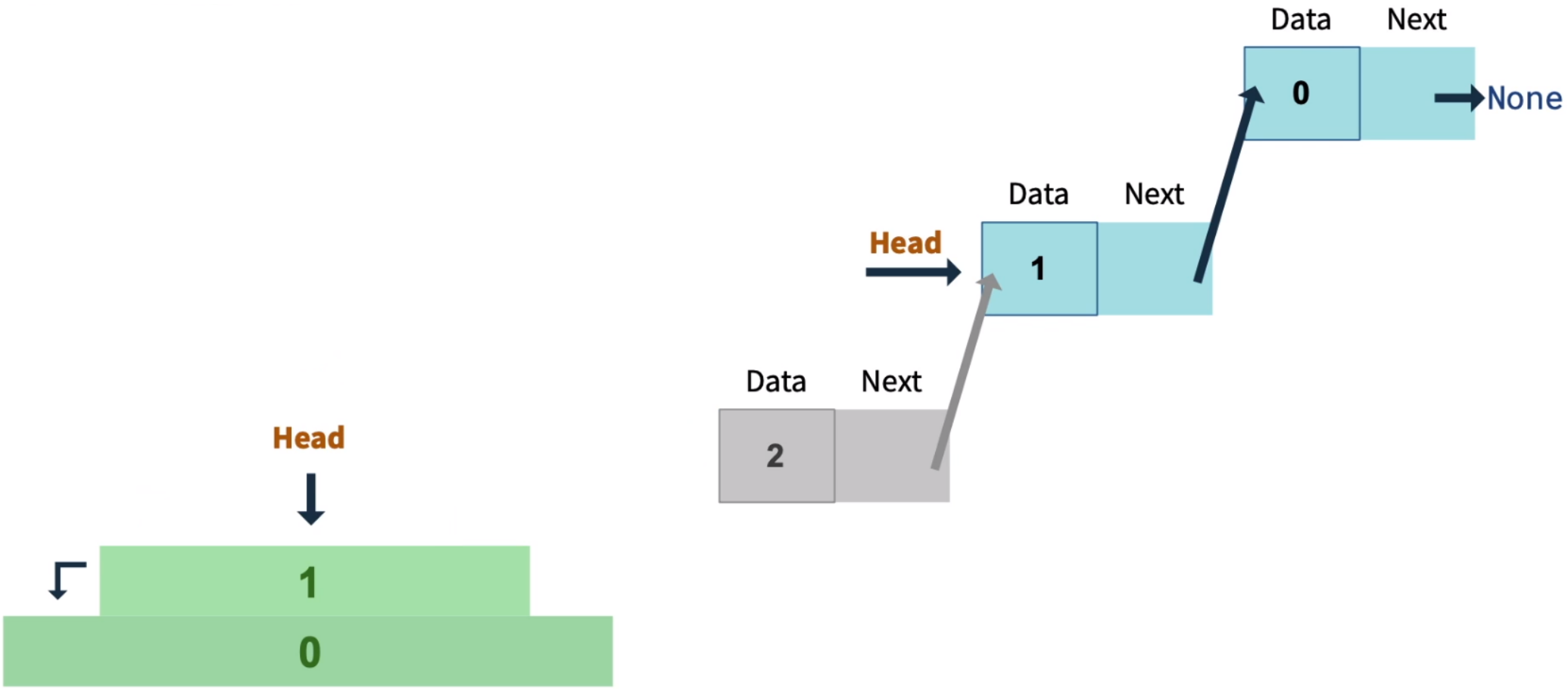
First Out

Using a Linked List to create a Stack

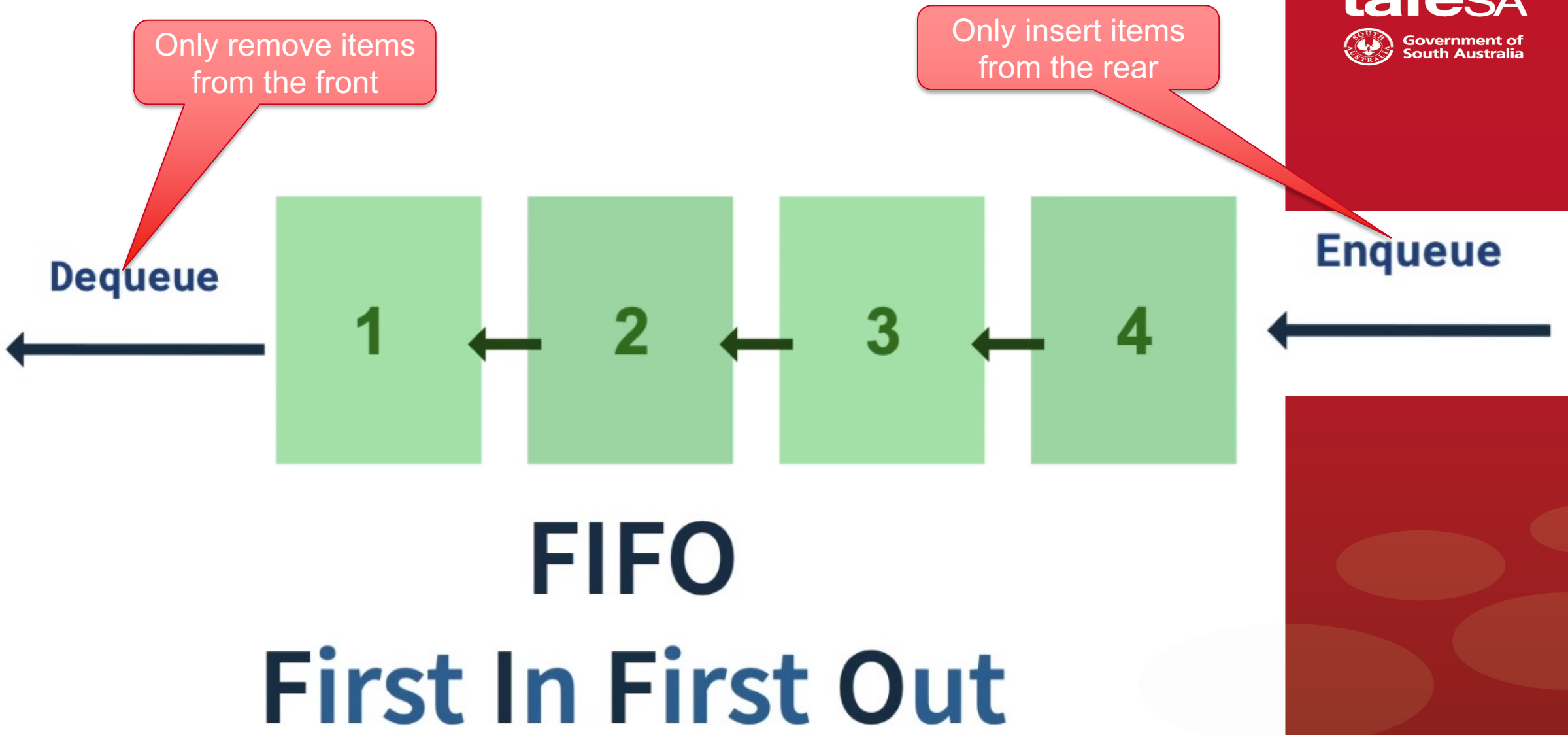




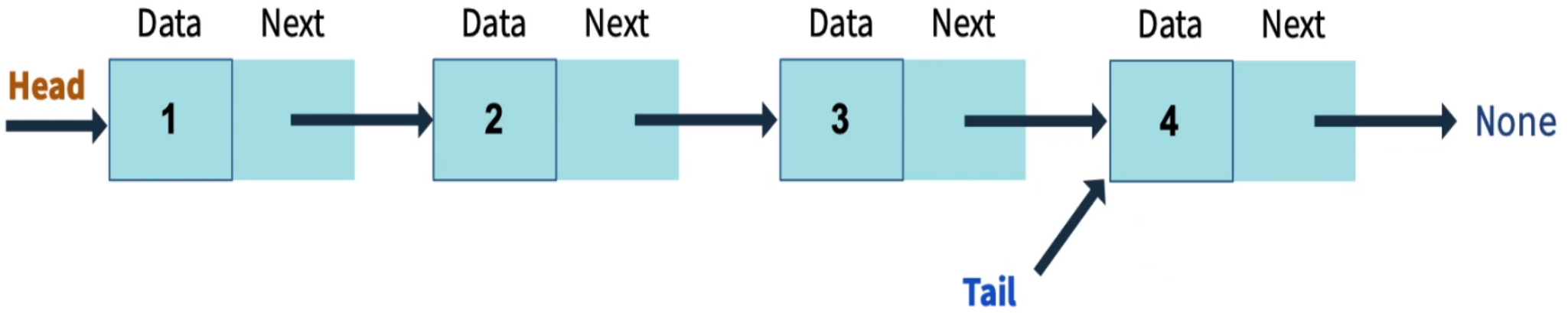
Removing the Top Node od Stack



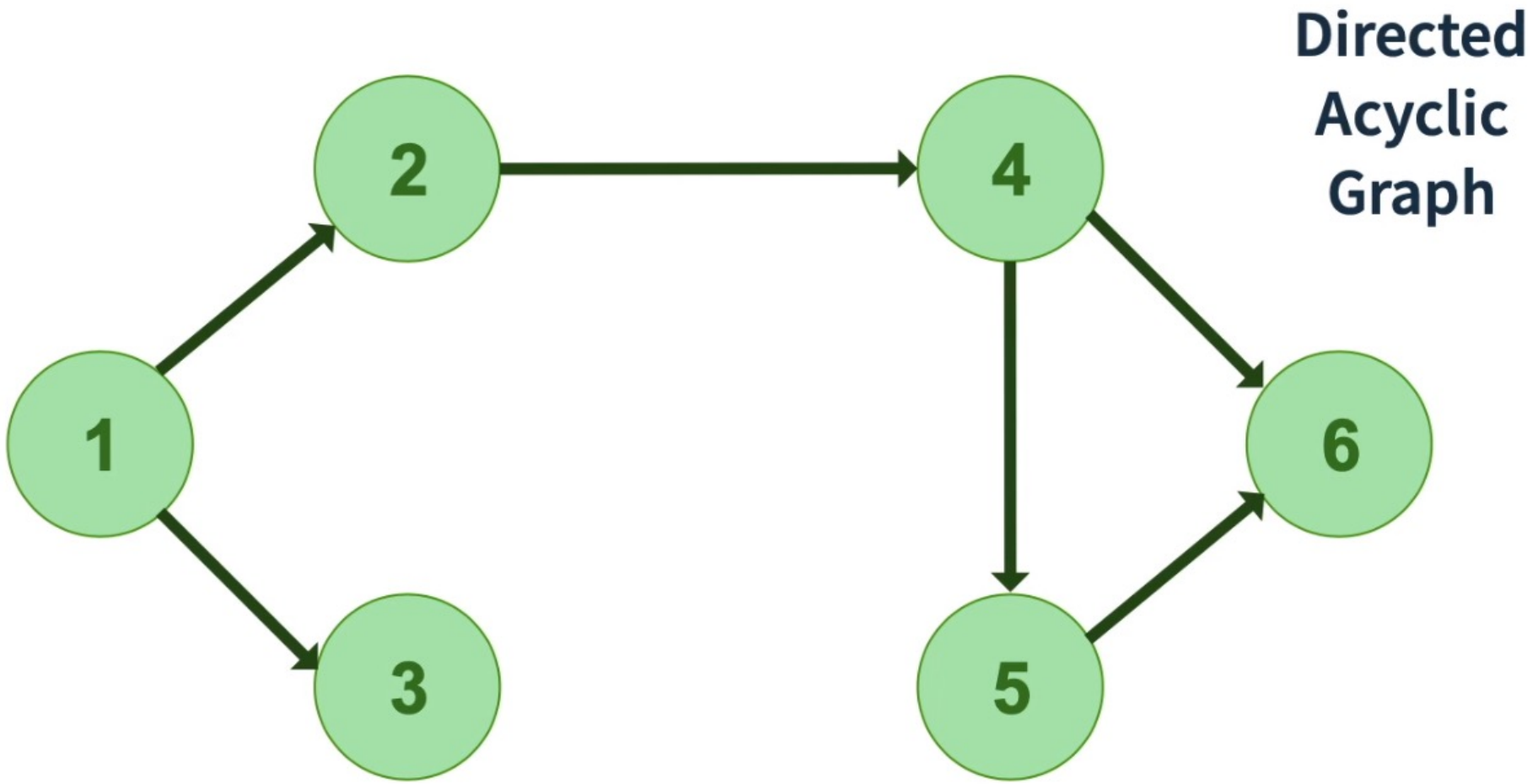
The Queue


Only remove items from the front
Only insert items from the rear
Dequeue
1
2
3
4
Enqueue
FIFO
First In First Out

Implementing Queue as a Linked List



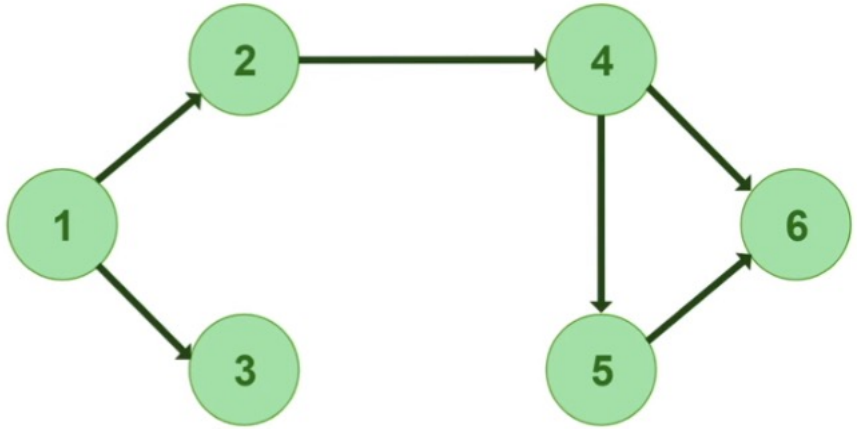
Graphs



Representing Graph as a Ajacency List (Dictionary)

Adjacency List

Vertex	Linked List of Vertices
1	2 -> 3 -> None
2	4 -> None
3	None
4	5 -> 6 -> None
5	6 -> None
6	None



Using collections.deque to create a Linked List(LL)

- deque (pronounced ‘deck’) is a specific object in the collections module that’s implements a Linked List structure
- Can be used to access, insert or remove elements from the beginning or end of a LL with a constant O(1) performance
- To cteate an empty LL:

```
>>> from collections import deque
>>> deque()
deque([])
```

- Can populate the LL using an iterable object as input:

```
>>> deque(['a','b','c'])
deque(['a', 'b', 'c'])
>>> deque('abc')
deque(['a', 'b', 'c'])
>>> deque([{'data':'a'},{'data':'b'}])
deque([{'data': 'a'}, {'data': 'b'}])
```

Adding and Removing items from a deque

- Use append() and pop() to add and remove elements from the right hand side of the LL

```
>>> llist = deque("abcde")
>>> llist
deque(['a', 'b', 'c', 'd', 'e'])

>>> llist.append("f")
>>> llist
deque(['a', 'b', 'c', 'd', 'e', 'f'])

>>> llist.pop()
'f'

>>> llist
deque(['a', 'b', 'c', 'd', 'e'])
```

- Use appendleft() and popleft() to add and remove elements from the left side or head of the LL

```
>>> llist.appendleft("z")
>>> llist
deque(['z', 'a', 'b', 'c', 'd', 'e'])

>>> llist.popleft()
'z'

>>> llist
deque(['a', 'b', 'c', 'd', 'e'])
```

Implementing a Queue using deque

- Use a Queue when you want a FIFO management of items in the queue.

- Example, implementing a fair seating of guests in a fully booked restaurant

```
>>> from collections import deque
>>> queue = deque()
>>> queue.append("Dale")
>>> queue.append("Nadil")
>>> queue.append("Julie")
>>> queue.append("Roberto")
>>> queue.append("KT")
>>> queue
deque(['Dale', 'Nadil', 'Julie', 'Roberto', 'KT'])
```

- To remove items from the queue, start with the item at the head (from the left) using the popleft() method.

```
>>> queue.popleft()
'Dale'
>>> queue.popleft()
'Nadil'
>>> # The queue after removing first two customers
    deque(['Julie', 'Roberto', 'KT'])
```

Implementing a Stack using deque

- Stacks manages elements in a FILO approach – meaning the last element added is the first to be removed
- Example,creating a web browser page history to store every page a user visits to facilitate page navigation from last visited to first visited :

```
>>> from collections import deque
>>> history = deque()
>>> history.appendleft("https://www.tafesa.edu.au/")
>>> history.appendleft("https://www.tafesa.edu.au/about-tafesa")
>>> history.appendleft("https://www.tafesa.edu.au/courses/information-technology")
>>> history
deque(['https://www.tafesa.edu.au/courses/information-technology', 'https://www.tafesa.edu.au/about-tafesa', 'ht
```

- To go back to the home page, do the following:

```
>>> history.popleft()
'https://www.tafesa.edu.au/courses/information-technology'
>>> history.popleft()
'https://www.tafesa.edu.au/about-tafesa'
>>> history
deque(['https://www.tafesa.edu.au/'])
```

Next Week….

- Binary Trees
- Searching Algorithms
- Sorting Algorithms