001 Stack Intro.mp4

In [ ]:

002 Stack Constructor.mp4

In [5]:
```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class Stack:
    def __init__(self, value):
        new_node = Node(value)
        self.top = new_node
        self.height = 1

    def print_stack(self):
        temp = self.top
        while temp!= None:
            print(temp.value)
            temp = temp.next
my_stack = Stack(10)

# print(my_stack.print_stack())

my_stack.print_stack()
```

```
10
```

003 Stack Push.mp4

In [7]:
```
class Node:
```

```python
    def __init__(self,value):
        self.value = value
        self.next = None

class Stack:
    def __init__(self, value):
        new_node = Node(value)
        self.top = new_node
        self.height = 1

    def print_stack(self):
        temp = self.top
        while temp!= None:
            print(temp.value)
            temp = temp.next

    # this is similar to prepend

    def push(self, value):
        new_node = Node(value)
        if self.height == 0 :
            self.top = new_node
        else:
            new_node.next = self.top
            self.top = new_node
        self.height +=1

my_stack = Stack(10)

my_stack.push(20)
my_stack.print_stack()

print(f'-------------Adding from the top-----')

my_stack.push(30)
my_stack.print_stack()
```

```
20
10
-------------Adding from the top-----
30
20
10
```

004 Stack Pop.mp4

In [8]:
```
class Node:
    def __init__(self,value):
        self.value = value
        self.next = None

class Stack:
    def __init__(self, value):
        new_node = Node(value)
        self.top = new_node
        self.height = 1

    def print_stack(self):
        temp = self.top
        while temp!= None:
            print(temp.value)
            temp = temp.next

    # this is similar to prepend

    def push(self, value):
        new_node = Node(value)
        if self.height == 0 :
            self.top = new_node
        else:
            new_node.next = self.top
            self.top = new_node
        self.height +=1

    def pop(self):
        if self.height == 0 :
```

```
            return None
        else:
            temp = self.top
            self.top = self.top.next
            temp.next = None
        self.height -=1
        return temp

my_stack = Stack(10)

my_stack.push(20)
my_stack.print_stack()

print(f'-------------Adding from the top-----')

my_stack.push(30)
my_stack.print_stack()

print(f'------------------------------------')
print(f'-------------Popping from the top-----')

my_stack.pop()
my_stack.print_stack()

print(f'------------------------------------')

print(my_stack.pop().value )
print(f'-------------Popping from the top-----')

my_stack.print_stack()
```

```
20
10
-------------Adding from the top-----
30
20
10
------------------------------------
-------------Popping from the top-----
20
10
------------------------------------
20
-------------Popping from the top-----
10
```

Sure! Here's a clean comparison table of the **main ways to implement a stack**, including performance, size limitations, and typical use cases:

📊 Stack Implementation Methods

Implementation	Push Time	Pop Time	Size Limit	Memory Usage	Typical Use Case	Notes
Singly Linked List	O(1)	O(1)	Dynamic	More (node + pointer)	Custom data structures, academic use	Good for dynamic, flexible stacks
Array (low-level)	O(1)	O(1)	Fixed	Less	Embedded systems, C/C++ stacks	Needs resizing logic for dynamic behavior
Python `list`	O(1)*	O(1)*	Dynamic	Efficient	Most general Python usage	Fast if used from the end with `append/pop()`
`collections.deque`	O(1)	O(1)	Dynamic	Efficient	When frequent insertions/removals needed	Faster for large-scale or multi-end operations