Stacks are containers where objects can be inserted and removed following the LIFO principle (Last In, First Out). A stack can only hold elements of the same data type. The main operations (with their time complexities) that can be performed on a stack are as follows:

- Push (Insert) -> O(1)
- Pop (Remove) -> O(1)
- Peek (Retrieve the top element) -> O(1)

Stacks can be implemented with the help of linked lists and arrays. Using linked lists, a stack can be implemented as per below:

In [25]:

```
class Node():
    def __init__(self, data):
        self.data = data
        self.next = None
# Because linked lists are composed of nodes, we start by creating the Node class, which will contain the data and

class Stack():
    def __init__(self):
        self.top = None
        self.bottom = None
        self.length = 0
# After that, we create the Stack class where the constructor will have the top pointer that will refer to the ele
# Then, we create the methods associated with a stack(peek, push and pop):

    def peek(self):
        if self.top is None:
            return None
        return self.top.data

# This function will retrieve the top element of the stack without removing it.
# The time complexity of this action is O(1), because we it only returns what the top pointer is referring to.

    def push(self, data):
        new_node = Node(data)
        if self.top == None:
            self.top = new_node
            self.bottom = new_node
        else:
            new_node.next = self.top
```

```
            self.top = new_node
        self.length += 1
# Next up, we create push(), which inserts an element at the top of the stack. Just like peek(), the time complexi
# If the stack is empty, the method will set both top and bottom pointer to refer the new_node.
# If not, the node next to new_node(which was pointing at None) refer to the current top pointer and, only after t
# At the end, the method updates the stack's length by 1.

    def pop(self):
        if self.top == None:
            print('Oops! This stack is empty.')
        else:
            self.top = self.top.next
            self.length -= 1
            if (self.length == 0):
                self.bottom = None
                return 'Stack is now empty.'
# Now, we build pop(), which is going to remove the top element from the stack. The time complexity is also O(1).
# If the stack is already empty, the method outputs a message.
# Otherwise, it makes the top pointer refer the element that was next to the 'popped' top pointer and decrease the
# Also, if there was only one element in the stack and it gets 'popped', the method will set the bottom pointer to

    def get_stack(self):
        if self.top == None:
            print('Oops! This stack is empty.')
        else:
            current_pointer = self.top
            while (current_pointer != None):
                print(current_pointer.data)
                current_pointer = current_pointer.next
# Finally, we build a method that will output all the elements in the stack from top to bottom. As this method tra
# If the stack is empty, the method returns a message.
```

Now, all that is left to do is test:

In [26]:

```
# Building a stack and ensuring it is empty.
new_wall = Stack()

print(new_wall.peek())
```

```
None
```

In [27]:
```
# Adding elements to the newly created stack.
new_wall.push('Blue bricks')
new_wall.push('Purple brick')
new_wall.push('Red bricks')
```

In [28]:
```
# Retrieve all elements of stack.
new_wall.get_stack()
```

```
Red bricks
Purple brick
Blue bricks
```

In [29]:
```
# Location of the top element of stack in memory.
print(new_wall.top)
```

```
<__main__.Node object at 0x7fe2e08efa00>
```

In [30]:
```
# Retrieving data of the top element of stack.
print(new_wall.top.data)
```

```
Red bricks
```

In [31]:
```
# Location of the bottom element of stack in memory.
print(new_wall.bottom)
```

```
<__main__.Node object at 0x7fe2e08efd00>
```

In [32]:
```
# Retrieving data of the bottom element of stack.
print(new_wall.bottom.data)
```

```
Blue bricks
```

In [33]:
```
# Location of stack in memory.
print(new_wall)
```

```
<__main__.Stack object at 0x7fe2e08efbe0>
```

In [34]:

```
# Removing top element of stack.
new_wall.pop()
```

In [35]:

```
# Retrieve all current elements of stack.
new_wall.get_stack()
```

```
Purple brick
Blue bricks
```

In [36]:

```
# Looking at the elements on top of the stack.
print(new_wall.peek())
```

```
Purple brick
```

In [37]:

```
# Emptying the stack and retrieving it.
new_wall.pop()
new_wall.pop()
new_wall.get_stack()
```

```
Oops! This stack is empty.
```