VNUHCM - University of Science
fit@hcmus

Fundamentals of Programming for Artificial Intelligence

Session 08
Doubly Linked List & Stack/Queue

Instructors:
Dr. Lê Thanh Tùng
Dr. Nguyễn Tiến Huy

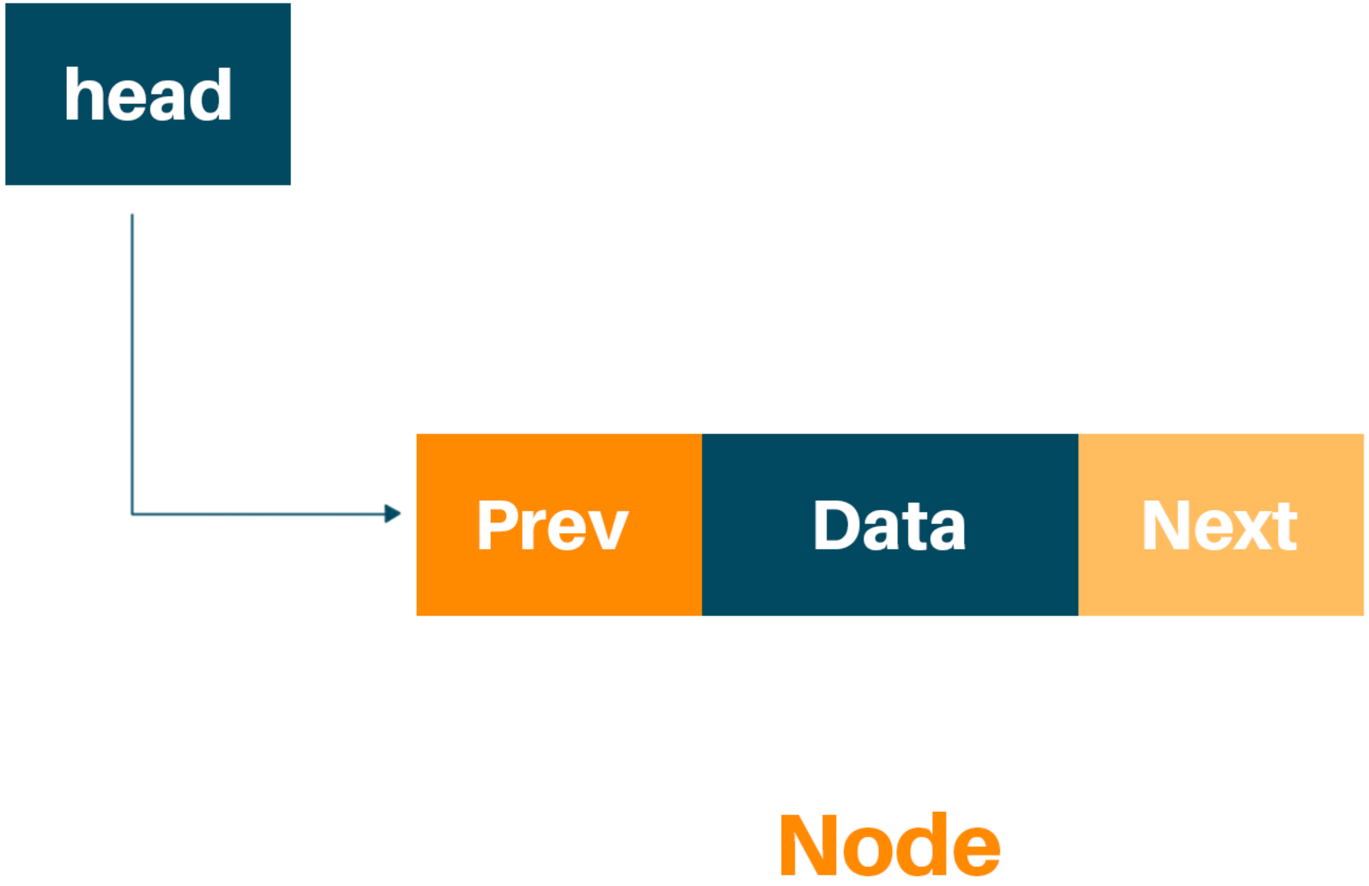
Content

1. Doubly Linked List
2. Circular Linked List
3. Stack
4. Queue

1. Doubly Linked List

Doubly Linked List

- Each node has 2 pointers:
  - 1 pointer to its successor (next pointer)
  - 1 pointer to its predecessor (previous pointer)



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

Doubly Linked List

- Let’s consider the structure of one node in Doubly Linked List

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

Doubly Linked List

- Let's consider the structure of one node in Doubly Linked List

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

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

Doubly Linked List

- Let's consider the operations in Doubly Linked List
  - Inserting Items to the Start
  - Inserting Items at the End
  - Deleting Elements from the Start
  - Deleting Elements from the End
  - Traversing the Linked List

- Function to insert a new node at the front of the list

```
def add_head(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
    else:
        new_node.next = self.head
        self.head.prev = new_node
        self.head = new_node
```

Doubly Linked List

- Function to insert a new node at the end of the list

```
def insert_last(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
    else:
        temp = self.head
        while temp.next is not None:
            temp = temp.next
        temp.next = new_node
        new_node.prev = temp
```

Doubly Linked List

- Function to delete a new node from the start of the list

```
def delete_head(self):
    if self.head is None:
        return
    if self.head.next is None:
        self.head = None
    else:
        self.head = self.head.next
        self.head.prev = None
```

- Function to delete a new node from the end of the list

```
def delete_last(self):
    if self.head is None:
        return
    if self.head.next is None:
        self.head = None
    else:
        temp = self.head
        while temp.next is not None:
            temp = temp.next
        temp.prev.next = None
```

- Let's think about
  - Write a function to check whether a linked list is palindrome or not

Linked List


```python
def is_palindrome(self):
    # Empty list is considered a palindrome
    if self.head is None:
        return True

    # Initialize pointers for comparison
    left = self.head
    right = self.head

    # Move right pointer
    # to the end of the list
    while right.next:
        right = right.next

    # Compare the nodes from start (left)
    # and end (right)
    while left != right and left.prev != right:
        if left.item != right.item:
            return False
        left = left.next
        right = right.prev

    return True
```

Doubly Linked List – Analysis

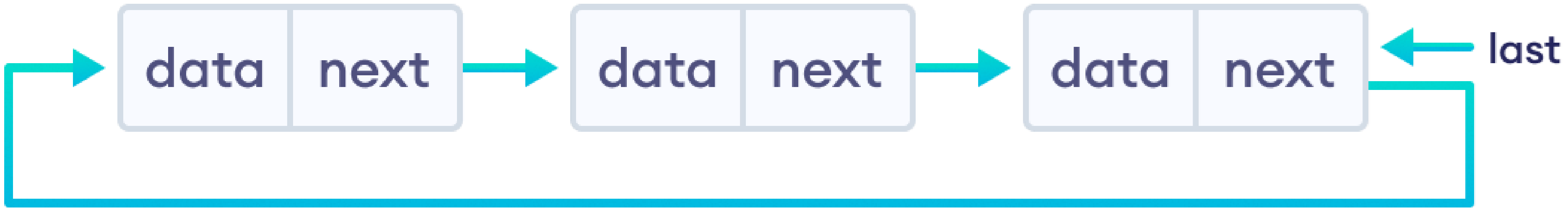
- Advantage:
  - Adding/removing are simpler and potentially more efficient for nodes other than first nodes
- Disadvantage:
  - Require changing more links than singly linked list when adding/removing a node

Doubly Linked List – Analysis

Doubly Linked List	Singly Linked List
Traversal: both forward and backward directions	Traversal: sequentially, forward direction
Deletion: more efficient if a pointer to the node to be deleted is given	Deletion: Need to reveal the previous and current node for deleting
Insertion: quickly insert a new node before a given node	Insertion: must control the surrounding nodes
Requires extra space for a previous pointer	Only a next pointer
All operations require an extra pointer previous to be maintained	No need

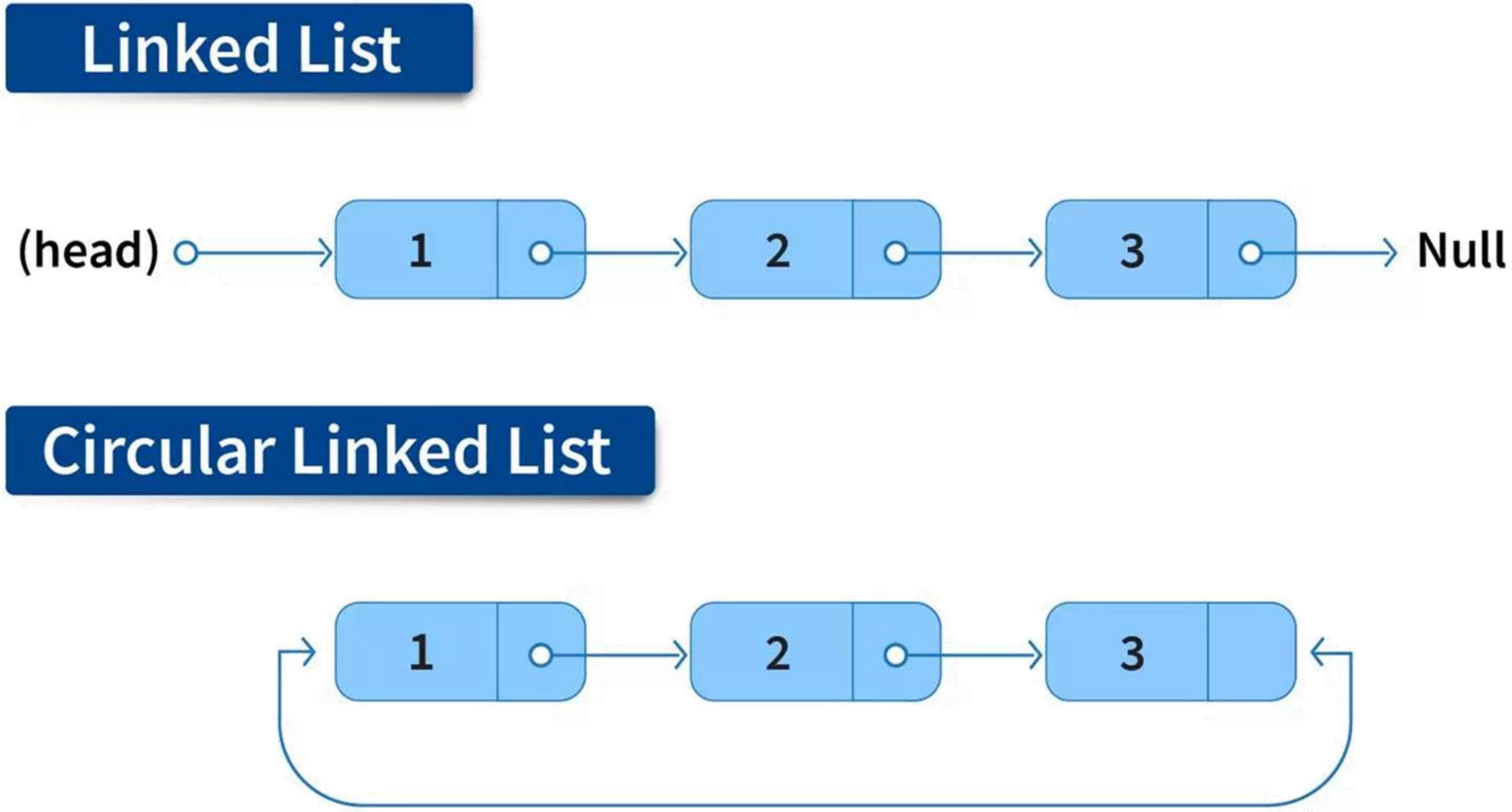
2. Circular Linked List

- A Circular Linked List is a linked list where the last node points back to the head, forming a circle



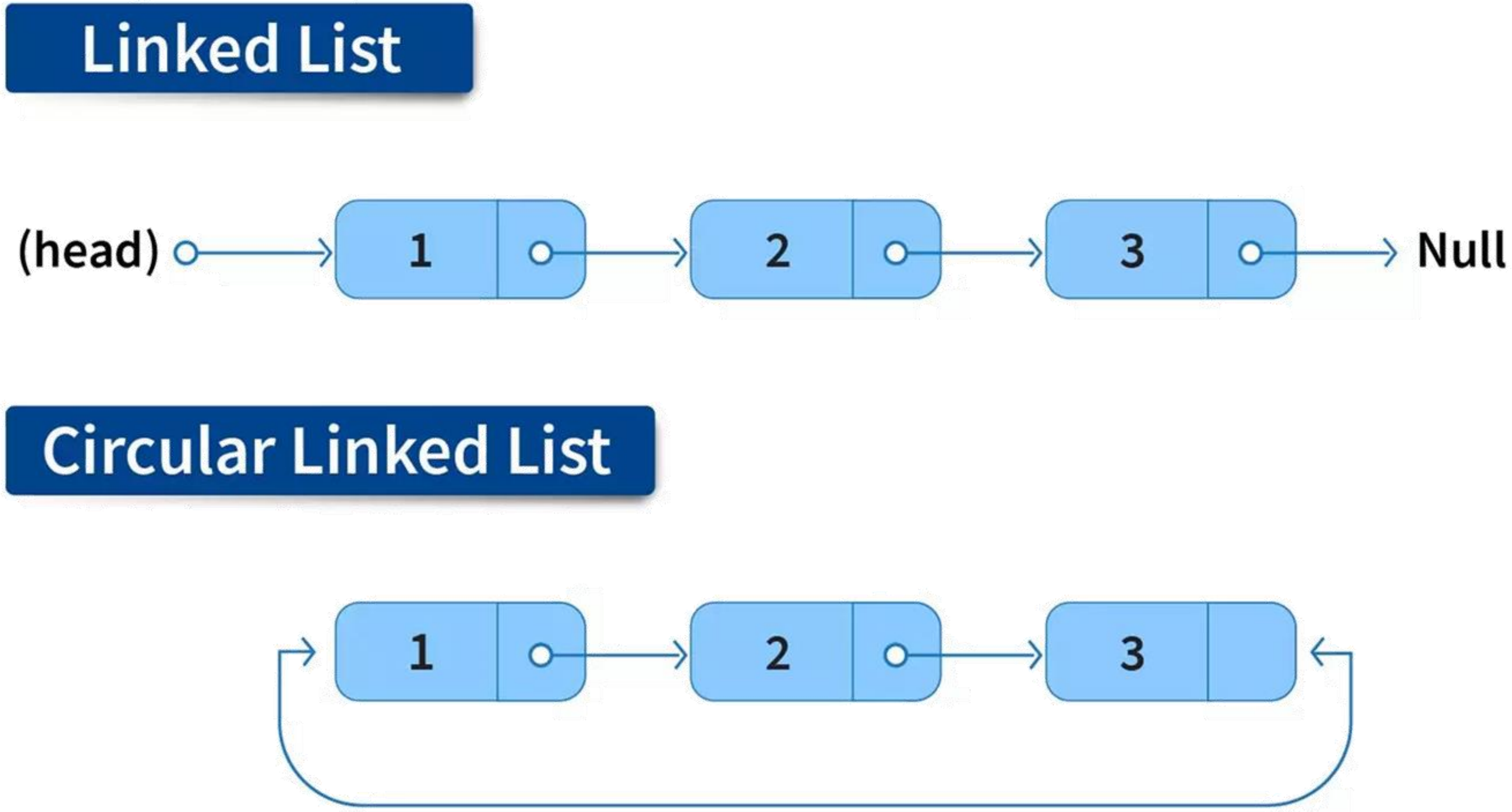
- The address of the last node consists of the address of the first node
- Circular linked lists can be singly linked or doubly linked list

- Nodes form a ring:
  - The first element point the next element, the last element points to the first element.
  - There is no NULL at the end!
  - Can be used to traverse the same list again and again



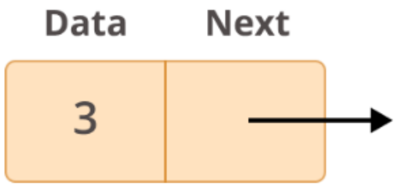
Circular Linked List

- A pointer to any node serves as a handle to the whole list
- Circular linked list may be used to represent:
  - Arrays that are naturally circular, e.g. the corners of a polygon
  - A pool of buffers that are used and released in First in, first out order

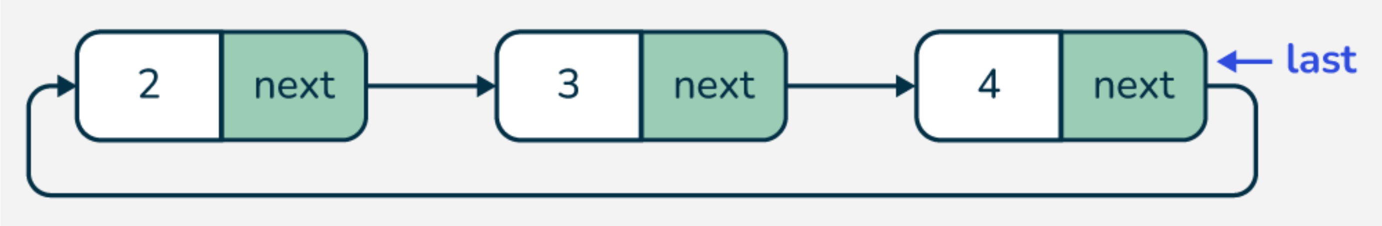


Node in Circular Linked List

- A node in a linked list contains one or more members that represent data
- Each node also contains (at least) a link to another node



Node




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

```python
class Circular_LinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0
```

Circular Linked List

- Let's consider the operations in Circular Linked List
  - Traversing the Linked List
  - Inserting Items to the Start
  - Inserting Items at the End
  - Deleting Elements from the Start
  - Deleting Elements from the End

- Traversing the Linked List

```python
def traverse(self):
    if self.head is None:
        print("The list is empty.")
        return
    current = self.head

    while True:
        print(current.data, end="->")
        current = current.next
        # Circular check
        if current == self.head:
            break
    print("None")
```

Circular Linked List

- Inserting Items to the Start

```python
def insert_at_start(self, value):
    new_node = Node(value)
    if self.head is None:
        # When list is empty,
        # new node becomes head and tail,
        # and points to itself
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
    else:
        new_node.next = self.head
        self.tail.next = new_node
        self.head = new_node
    self.size += 1
```

- Inserting Items at the End

```
def insert_at_end(self, value):
    new_node = Node(value)
    if self.head is None:
        # If the list is empty, new node is both head and tail
        self.head = new_node
        self.tail = new_node
        new_node.next = new_node
    else:
        new_node.next = self.head
        self.tail.next = new_node
        self.tail = new_node
    self.size += 1
```

- Deleting Elements from the Start

```python
def delete_from_start(self):
    if self.head is None:
        print("The list is empty. Cannot delete.")
        return

    if self.head == self.tail:
        # Only one node in the list
        self.head = None
        self.tail = None
    else:
        self.tail.next = self.head.next  # Update the last node's next pointer
        self.head = self.head.next  # Move head to the next node
    self.size -= 1
```

- Deleting Elements from the End

```python
def delete_from_end(self):
    if self.head is None:
        print("The list is empty. Cannot delete.")
        return

    if self.head == self.tail:
        # Only one node in the list
        self.head = None
        self.tail = None
    else:
        current = self.head
        while current.next != self.tail:
            current = current.next
        current.next = self.head  # Update the second-to-last node's next to point to head
        self.tail = current  # Update tail to second-to-last node
    self.size -= 1
```

- Advantage:
  - Any node can be a starting point. We can traverse the whole list from any point
  - Useful for applications to repeatedly go around the list:
    - Applications in PC
    - Circular Queue

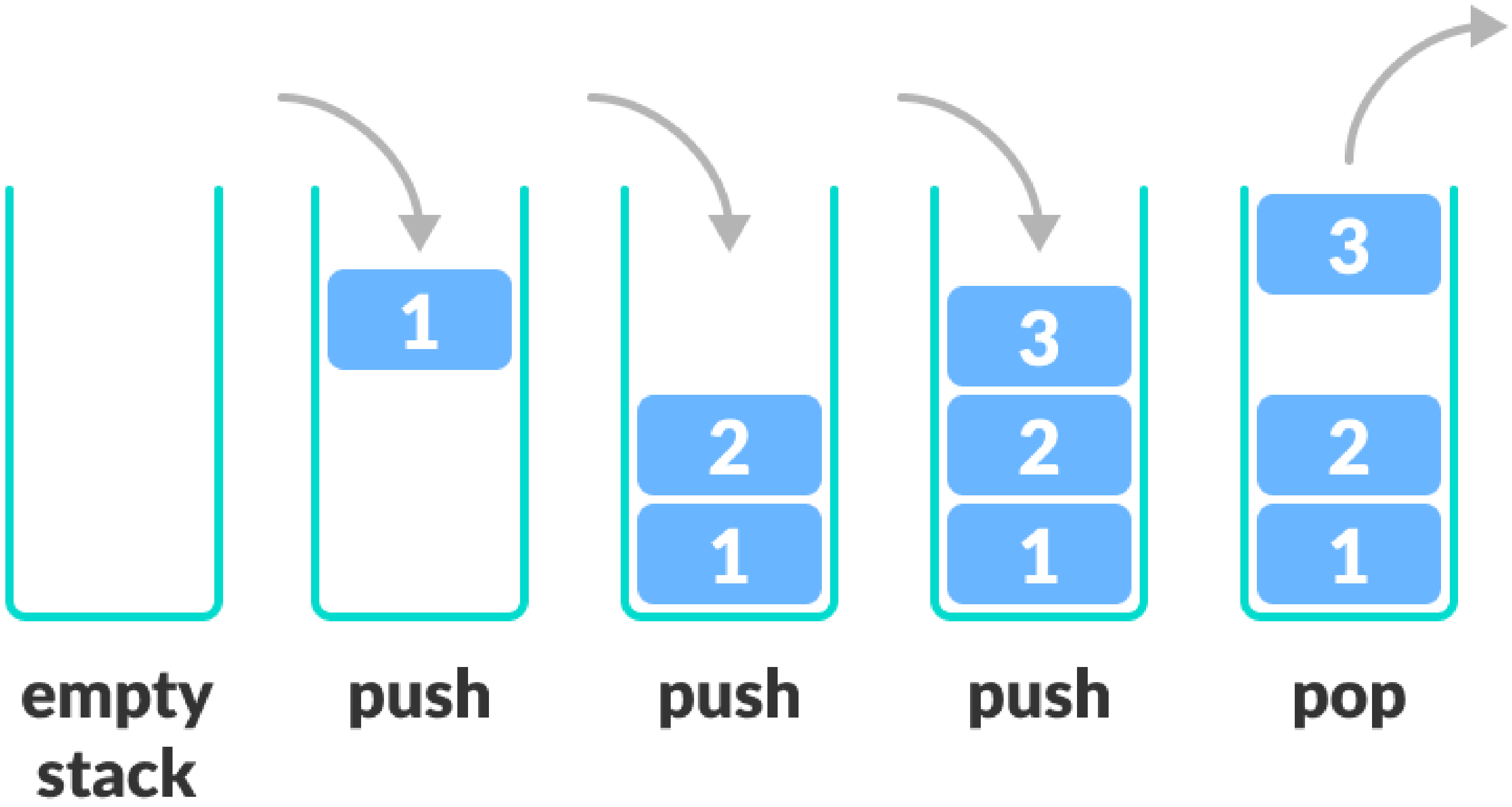
Circular Linked List – Analysis

- Disadvantage:
  - Finding the end of a list is more difficult (no NONE to mark the beginning / end)
  - Add at beginning could be expensive to search for the last node (depending on the implementation)

3. Stack

Stack

- A stack is a linear data structure that follows the principle of **Last In First Out (LIFO)**
- That is, elements can be added to and removed from a stack only at the top



- The following are the basic operations served by stacks.
  - **push()**: Adds an element to the top of the stack
  - **pop()**: Removes the topmost element from the stack.
  - **isEmpty()**: Checks whether the stack is empty.
  - **peek()/top()**: Displays the topmost element of the stack.
  - **size()**: returns the size of stack
  - **isFull()**: Checks whether the stack is full.

Stack

fit@hcmus

- There are various ways from which a stack can be implemented in Python.
- Stack in Python can be implemented using the following ways:
  - List
  - Singly linked list
  - Collections.deque
  - queue.LifoQueue

Implement a Stack

- Using a List :

```
class Stack:
    def __init__(self, max_size = None):
        self.__data__ = []
        self.__length__ = 0
        if max_size is None:
            self.max_size = 1000
        else:
            self.max_size = max_size
```

Implement a Stack

- We can take advantage of the operations in list with the “strictly defined” procedure of stack

```
def peek(self):
    if self.isEmpty():
        print("Stack is empty. No top item.")
        return None
    return self.__data__[-1]

def size(self):
    return self.__length__
```

Implement a Stack

- We can take advantage of the operations in list with the "strictly defined" procedure of stack

```python
def isEmpty(self):
    return self.__length__ == 0

def isFull(self):
    if self.max_size is None:
        return False
    return self.__length__ >= self.max_size
```

Implement a Stack

- We can take advantage of the operations in list with the “strictly defined” procedure of stack

```python
def push(self, item):
    if self.max_size is not None:
        if self.__length__ >= self.max_size:
            print("Cannot push new item.")
            return
    self.__data__.append(item)
    self.__length__ += 1
```

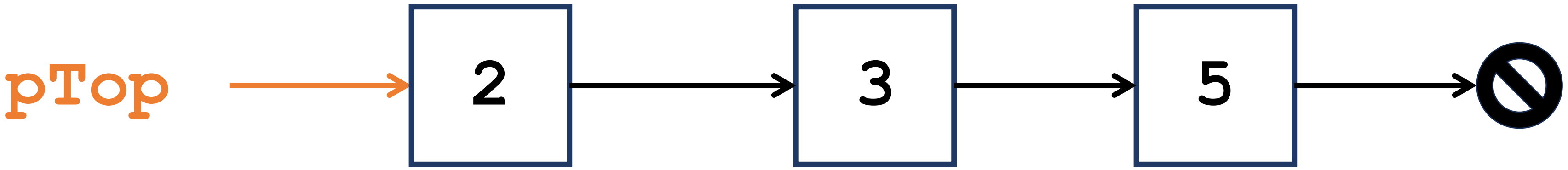
Implement a Stack

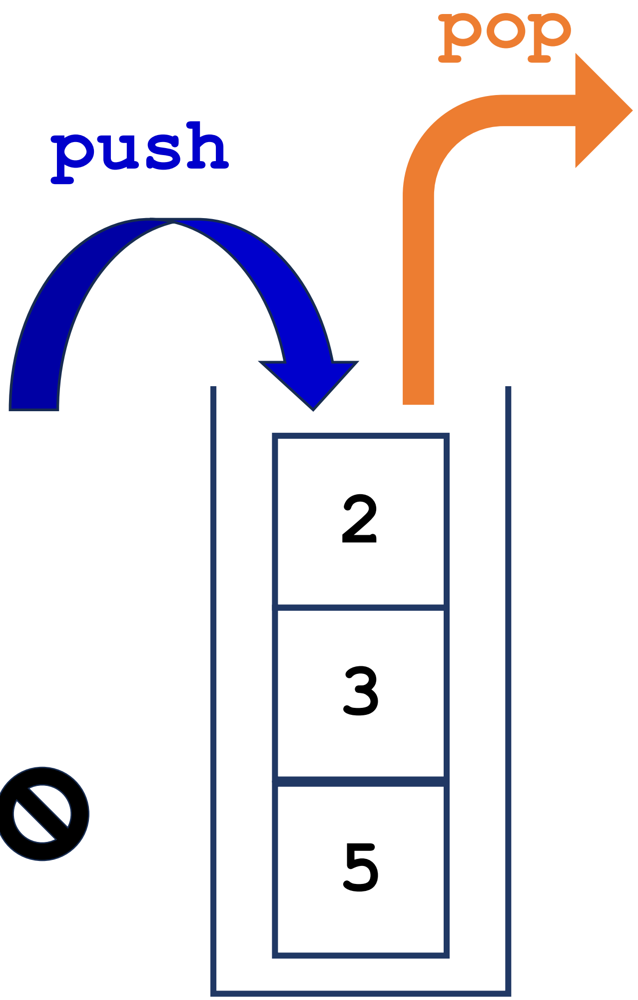
- We can take advantage of the operations in list with the “strictly defined” procedure of stack

```python
def pop(self):
    if self.isEmpty():
        print("Stack is empty. Cannot pop item.")
        return None
    self.__length__ -= 1
    return self.__data__.pop()
```

Implement a Stack

- Using a singly-linked list:
  - A pointer pTop points to the top of the stack
  - Empty stack: `pTop = NULL`

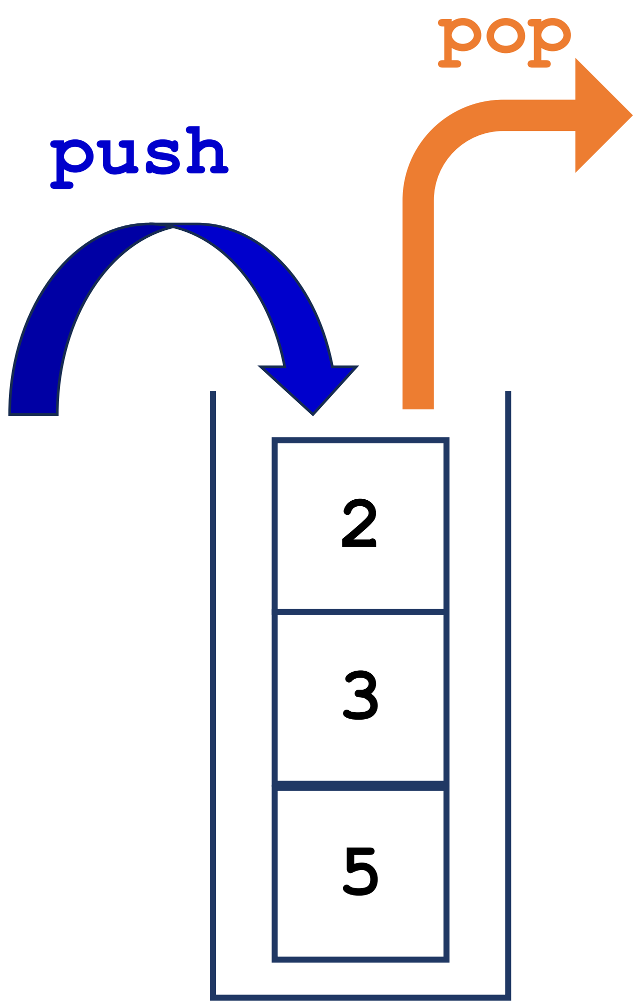




Implement a Stack

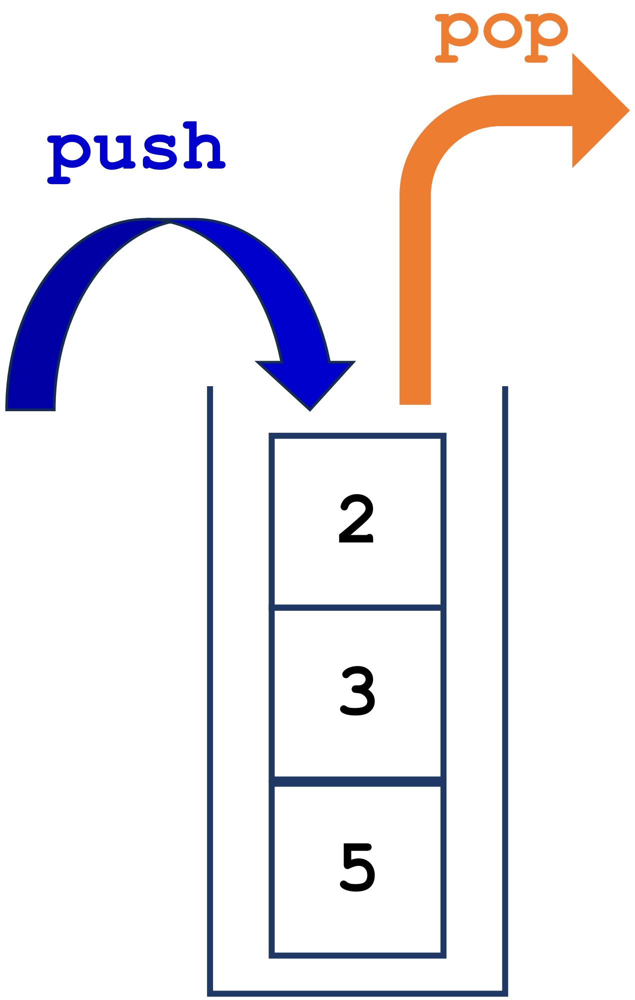
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Stack:
    def __init__(self, max_size=None):
        self.__data__ = None
        self.__length__ = 0
        if max_size is None:
            self.max_size = 1000
        else:
            self.max_size = max_size
```



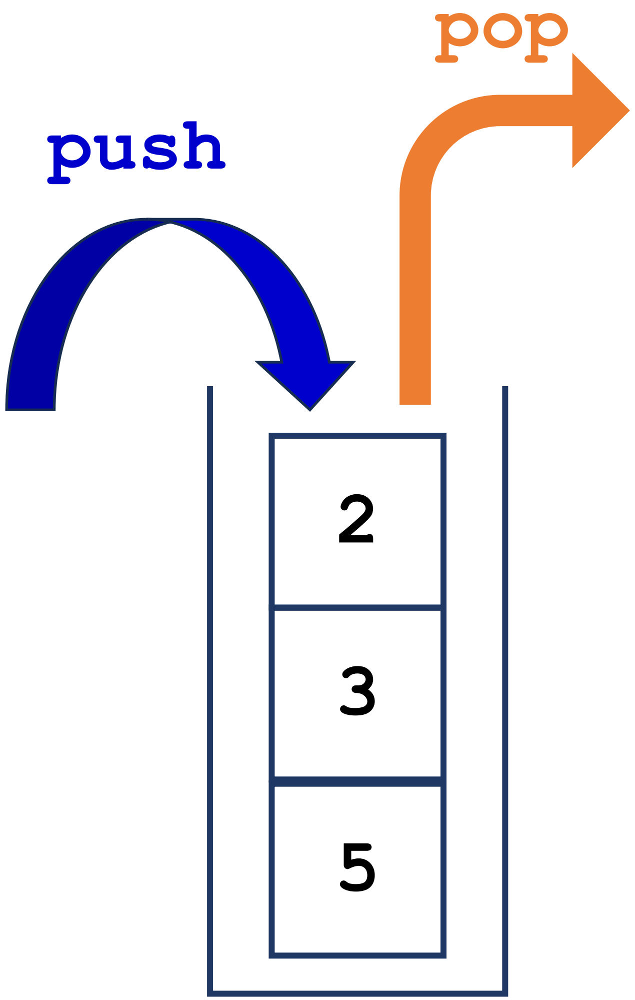
Implement a Stack

```python
def push(self, item):
    if self.isFull():
        print("Cannot push new item.")
        return
    new_node = Node(item)
    new_node.next = self.__data__
    self.__data__ = new_node
    self.__length__ += 1
```



Implement a Stack

```
def pop(self):
    if self.isEmpty():
        print("Cannot pop item.")
        return None
    popped_node = self.__data__
    self.__data__ = self.__data__.next
    self.__length__ -= 1
    return popped_node.data
```



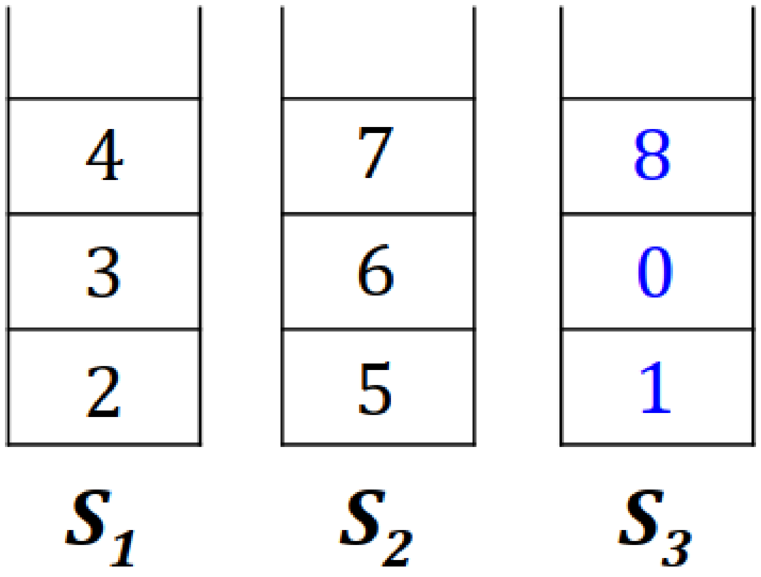
Stack Applications

- Delimiter Matching (part of any compiler)
  - C++: ( ) { } /* */ [ ]
  - Expression: `((a + b) * (c - d + e))`
  - Delimiters can be nested
  - Idea:
    - Opening Delimiters: When an opening delimiter ((, {, [, /*) is encountered, push it onto the stack
    - Closing Delimiters: When a closing delimiter (), },], */) is encountered, perform the following checks:
      - If the stack is empty, the expression is invalid.
      - Pop the top element from the stack. If the popped element does not match the corresponding opening delimiter for the closing delimiter encountered, the expression is invalid.

```python
def delimiter_matching(expression):
    stack = Stack()
    for char in expression:
        # Handle opening delimiters
        if char in '({[':
            stack.push(char)
        # Handle closing delimiters
        elif char in ')}]':
            if (stack.isEmpty() or
                not is_matching_pair(stack.pop(), char)):
                return False
    # If stack is not empty, there are unmatched opening delimiters
    return stack.isEmpty()
```

Stack Applications

- Adding two large number
  - Treat these numbers as strings of numerals, store the numbers corresponding to these numerals on 2 stacks
  - Perform addition by popping numbers from the stacks



- $\text{Pop}(S_1) + \text{Pop}(S_2) = 4+7 = 11$

→ Add 1 to S3

- $\text{Pop}(S_1) + \text{Pop}(S_2) = 3+6+1 = 10$

→ Add 0 to S3

- $\text{Pop}(S_1) + \text{Pop}(S_2) = 2+5+1 = 8$

→ Add 8 to S3

- S_1 S_2 are empty

→ $\text{Pop}(S_3)$ and get the result: 801

4. Queue

Queue

- A queue is a data structure that stores and retrieves elements in a first-in-first-out (or FIFO) manner

A single-lane one-way road

A queue of people waiting for a bus

Queue

- The following are the basic operations served by stacks.
  - **enqueue()**: adding an element to the end of the queue
  - **dequeue()**: removing an element from the front of the queue
  - **front()**: get the element at the front of the queue without removing it
  - **initialize()**: Creates an empty queue
  - **isEmpty()**: Check if the queue is empty
  - **isFull()**: Checks if the queue is full

Queue

- There are various ways to implement a queue in Python
- Python Queue can be implemented by the following ways
  - List
  - Linked List
  - collections.deque
  - queue.Queue

Implement a Queue

```
class Queue:
    def __init__(self, max_size=None):
        self.__data__ = []
        self.__length__ = 0
        if max_size is None:
            self.max_size = 1000
        else:
            self.max_size = max_size
```

Implement a Queue

```
def enqueue(self, item):
    if self.isFull():
        print("Cannot add new item.")
        return
    self.__data__.append(item)
    self.__length__ += 1
```

Implement a Queue


```python
def dequeue(self):
    if self.isEmpty():
        print("Cannot remove item.")
        return None
    self.__length__ -= 1
    return self.__data__.pop(0)
```

Implement a Queue

```python
def front(self):
    if self.isEmpty():
        print("Queue is empty. No front item.")
        return None
    return self.__data__[0]
```

Implement a Queue

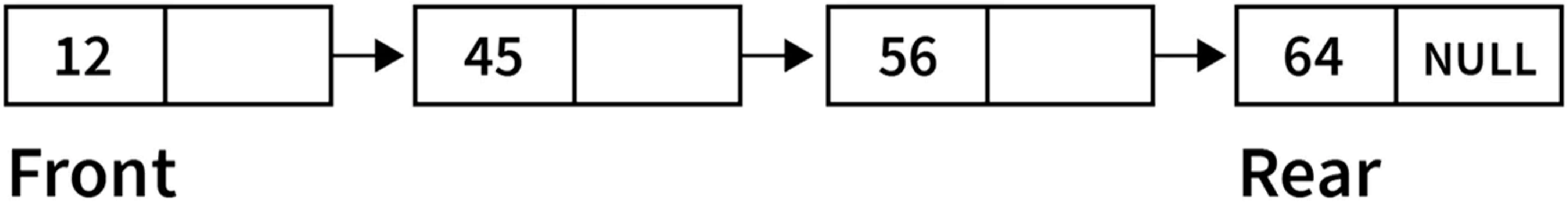

```
def isEmpty(self):
    return self.__length__ == 0

def isFull(self):
    return self.__length__ >= self.max_size

def size(self):
    return self.__length__
```

Implement a Queue

- Using Linked List:



Implement a Queue


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

Implement a Queue

```
class Queue:
    def __init__(self, max_size=None):
        self.front = None
        self.rear = None
        self.__length__ = 0
        if max_size is None:
            self.max_size = 1000
        else:
            self.max_size = max_size
```

Implement a Queue


```python
def enqueue(self, item):
    if self.isFull():
        print("Cannot add new item.")
        return
    new_node = Node(item)
    if self.rear is None:
        self.front = self.rear = new_node
    else:
        self.rear.next = new_node
        self.rear = new_node
    self.__length__ += 1
```

Implement a Queue


```
def dequeue(self):
    if self.isEmpty():
        print("Cannot remove item.")
        return None
    removed_node = self.front
    self.front = self.front.next
    if self.front is None:
        self.rear = None
    self.__length__ -= 1
    return removed_node.data
```

Implement a Queue


```python
def front_item(self):
    if self.isEmpty():
        print("Queue is empty. No front item.")
        return None
    return self.front.data
```

Queue Applications

- Task Scheduling
- Resource Allocation
- Message buffering
- Traffic Management
- Print jobs, procedures management
- Download jobs in browsers

- Reverse a Queue
- Objective: Implement a function in Python that reverses the elements of a queue. You may use a stack as an auxiliary data structure to hold the elements of the queue while you perform the reversal.

Approach

- Use a stack to hold the elements of the queue.
  - Dequeue all elements from the queue and push them onto the stack.
- Transfer elements back to the queue from the stack.
  - Pop all elements from the stack and enqueue them back to the queue.

Exercises

Step 1: Dequeue Elements to Stack

- While the queue is not empty:
  - Dequeue the front element from the queue.
  - Push the element onto the stack.

Step 2: Enqueue Elements from Stack

- While the stack is not empty:
  - Pop the top element from the stack.
  - Enqueue the element back to the queue.

Exercises

```python
def reverseQueue(queue):
    stack = Stack()

    # Step 1: Transfer elements from queue to stack
    while not queue.isEmpty():
        stack.push(queue.dequeue())

    # Step 2: Transfer elements from stack back to queue
    while not stack.isEmpty():
        queue.enqueue(stack.pop())
```

Exercises

- Sort Values in a Stack
- Objective: Write a program in Python that sorts the values in a stack in ascending order. You may use additional stacks but no other data structures like arrays, lists, etc.

Approach

- Use an additional stack to hold sorted elements.
  - Pop elements from the original stack and push them onto the auxiliary stack in sorted order.
- Transfer elements back to the original stack to maintain ascending order.

Exercises

Step 1: Sort Using Auxiliary Stack

- While the original stack is not empty:
  - Pop the top element from the original stack.
  - While the auxiliary stack is not empty and the top element of the auxiliary stack is greater than the popped element:
    - Pop elements from the auxiliary stack and push them back to the original stack.
  - Push the popped element onto the auxiliary stack.

Step 2: Transfer Back to Original Stack

- Pop elements from the auxiliary stack and push them back to the original stack.

```python
def sortStack(stack):
    aux_stack = Stack()

    # Step 1: Sort using auxiliary stack
    while not stack.isEmpty():
        temp = stack.pop()
        while (not aux_stack.isEmpty()
                and aux_stack.peek() > temp):
            stack.push(aux_stack.pop())
        aux_stack.push(temp)

    # Step 2: Transfer back to original stack
    while not aux_stack.isEmpty():
        stack.push(aux_stack.pop())
```

THANK YOU

for YOUR ATTENTION