```
#Implement unlimited size stack
class unlimitedstack:
  def __init__(self):
    self.stack=[]
  def push(self,ele):
    self.stack.append(ele)
    return len(self.stack)-1
  def pop(self):
    if not len(self.stack) == 0:
      return self.stack.pop()
    else:
      return len(self.stck == 0)
  def isemtpty(self):
    return len(self.stck == 0)
  def size(self):
    return len(self.stck == 0)
  def peek(self):
    if not self.is_empty():
      return self.stack[-1]
stack = unlimitedstack()
index_1 = stack.push(1)
index_2 = stack.push(2)
index_3 = stack.push(3)
index_4 = stack.pop()
assert index_1 == 0
assert index_2 == 1
assert index_3 == 2
assert index_4 == 3

print("All test cases passed!")
```

```
All test cases passed!
```

```
#Implement limited size Stack
class LimitedStack:
    def __init__(self, max_size):
        self.stack = []
        self.max_size = max_size

    def push(self, ele):
        if len(self.stack) >= self.max_size:
            raise OverflowError("Stack has reached its maximum size")
        self.stack.append(ele)
        return len(self.stack) - 1

    def pop(self):
        if len(self.stack) > 0:
            return self.stack.pop()
        else:
            raise IndexError("Pop from an empty stack")

    def is_empty(self):
        return len(self.stack) == 0

    def size(self):
        return len(self.stack)

    def peek(self):
        if not self.is_empty():
            return self.stack[-1]
        else:
            raise IndexError("Peek from an empty stack")

stack = LimitedStack(3)

index_1 = stack.push(1)
index_2 = stack.push(2)
index_3 = stack.push(3)
index_4 = stack.pop()

assert index_1 == 0
assert index_2 == 1
assert index_3 == 2
assert index_4 == 3
print("All test cases passed!")
```

```
All test cases passed!
```

```
#Reverse the content of file using Stack
def is_balanced(expression):
    stack = []
    matching_parentheses = {')': '(', '}': '{', ']': '['}
    for char in expression:
        if char in '({[':
            stack.append(char)
        elif char in ')}]':
            if not stack or stack.pop() != matching_parentheses[char]:
                return False
    return len(stack) == 0
if __name__ == "__main__":
    expressions = ["(a + b)", "(a + b]", "[(a + b) * c]", "((a + b) * c)", "([{}])", "([)])"]
    for expr in expressions:
        if is_balanced(expr):
          print(f"The expression '{expr}' is balanced.")
        else:
            print(f"The expression '{expr}' is not balanced.")
```

```
The expression '(a + b)' is balanced.
The expression '(a + b]' is not balanced.
The expression '[(a + b) * c]' is balanced.
The expression '((a + b) * c)' is balanced.
The expression '([{}])' is balanced.
The expression '([)])' is not balanced.
```

```
#Match the parentheses using Stack
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        else:
            raise IndexError("Pop from an empty stack")

    def is_empty(self):
        return len(self.stack) == 0

def reverse_file_content(file_path):
    stack = Stack()

    with open(file_path, 'r') as file:
        for line in file:
            stack.push(line.strip())

    reversed_lines = []
    while not stack.is_empty():
        reversed_lines.append(stack.pop())

    with open(file_path, 'w') as file:
        for line in reversed_lines:
            file.write(line + '\n')

file_path = 'user/documents/samplefile.txt'
reverse_file_content(file_path)
```

```
#Match the tags in HTML file using Stack
from html.parser import HTMLParser

class TagMatcher(HTMLParser):
    def __init__(self):
        super().__init__()
        self.stack = []
        self.is_balanced = True

    def handle_starttag(self, tag, attrs):
        self.stack.append(tag)

    def handle_endtag(self, tag):
        if not self.stack or self.stack.pop() != tag:
            self.is_balanced = False

    def match_tags(self, html_content):
        self.feed(html_content)
        return self.is_balanced and not self.stack

def check_html_tags(html_content):
    matcher = TagMatcher()
    if matcher.match_tags(html_content):
        print("The HTML tags are properly matched.")
    else:
        print("The HTML tags are not properly matched.")

if __name__ == "__main__":
    html_content = """
    <html>
        <head><title>Test Page</title></head>
        <body>
    <h1>Welcome to the Test Page</h1>
            <p>This is a <strong>simple</strong> HTML file.</p>
        </body>
    </html>
    """
    check_html_tags(html_content)
```

```
The HTML tags are properly matched.
```

```
#6. Implement a function with signature transfer(S,T). This function transfers all elements from Stack S to Stack T. The sequence of ele
class Stack:
    def __init__(self):
        self.stack=[]

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        else:
          raise IndexError("Pop from an empty stack")

    def is_empty(self):
        return len(self.stack) == 0
def transfer(S, T):
    auxiliary_stack = Stack()
    while not S.is_empty():
        auxiliary_stack.push(S.pop())
    while not auxiliary_stack.is_empty():
        T.push(auxiliary_stack.pop())
if __name__ == "__main__":
  S = Stack()
  T = Stack()
  S.push(1)
  S.push(2)
  S.push(3)
  transfer(S, T)
  print("Stack S after transfer:", S.stack)
  print("Stack T after transfer:", T.stack)
```

```
Stack S after transfer: []
Stack T after transfer: [1, 2, 3]
```

```
#Implement “Forward” and “Back” buttons of browser using Stacks. Elements need to be stored are URLs.
class BrowserHistory:
    def __init__(self):
        self.back_stack = []
        self.back_stack = []
        self.forward_stack = []
        self.current_url = None
    def visit(self, url):
        if self.current_url is not None:
            self.back_stack.append(self.current_url)
        self.current_url = url
        self.forward_stack.clear()


    def back(self):
        if self.back_stack:
            self.forward_stack.append(self.current_url)
            self.current_url = self.back_stack.pop()
        else:
            print("No more history to go back.")
    def forward(self):
        if self.forward_stack:
            self.back_stack.append(self.current_url)
            self.current_url = self.forward_stack.pop()
        else:
            print("No more forward history.")
    def current(self):
        return self.current_url
if __name__ == "__main__":
    browser = BrowserHistory()
    browser.visit("http://example.com")
    browser.visit("http://example.com/page1")
    browser.visit("http://example.com/page2")
    print("Current URL:", browser.current())
    browser.back()
    print("After going back, current URL:", browser.current())
    print("After going back, current URL:", browser.current())
    browser.forward()
    print("After going forward, current URL:", browser.current())
    browser.forward()
    print("After going forward, current URL:", browser.current())
    browser.forward()
```

```
Current URL: http://example.com/page2
After going back, current URL: http://example.com/page1
After going back, current URL: http://example.com/page1
After going forward, current URL: http://example.com/page2
No more forward history.
After going forward, current URL: http://example.com/page2
No more forward history.
```

```
#Implement “Simple Queue” using list data structure.
class Queue:
    def __init__(self):
        self.items = []
    def enqueue(self, item):
        self.items.append(item)
    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            raise IndexError("Queue is empty")
    def is_empty(self):
        return len(self.items) == 0
    def size(self):
        return len(self.items)
    def peek(self):
        if not self.is_empty():
            return self.items[0]
        else:
            raise IndexError("Queue is empty")
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.dequeue())
print(q.peek())
print(q.size())
q.dequeue()
q.dequeue()
print(q.is_empty())
```

```
1
2
```

```
2
True
```

```
#Modify Q1 such that Simple Queue can contain limited amount of elements.
class Queue:
    def __init__(self, max_size):
        self.items = []
        self.max_size = max_size
    def enqueue(self, item):
        if len(self.items) < self.max_size:
            self.items.append(item)
        else:
            raise IndexError("Queue is full")
    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            raise IndexError("Queue is empty")
    def is_empty(self):
        return len(self.items) == 0
    def is_full(self):
        return len(self.items) == self.max_size
    def size(self):
        return len(self.items)
    def peek(self):
        if not self.is_empty():
            return self.items[0]
        else:
            raise IndexError("Queue is empty")
q = Queue(3)
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.is_full())
try:
    q.enqueue(4)
except IndexError:
    print("Queue is full")

print(q.dequeue())
print(q.size())
q.enqueue(4)
print(q.size())
```

```
True
Queue is full
1
2
3
```

```
#Implement “FlexiQueue” with capacity to expand and shrunk based on elements to be added or deleted.
class FlexiQueue:
    def __init__(self, initial_capacity=10):
        self.items = [None] * initial_capacity
        self.front = 0
        self.rear = 0
        self.size = 0
        self.capacity = initial_capacity
    def enqueue(self, item):
        if self.size == self.capacity:
            self._resize(self.capacity * 2)
        self.items[self.rear] = item
        self.rear = (self.rear + 1) % self.capacity
        self.size += 1
    def dequeue(self):
        if self.size == 0:
            raise IndexError("Queue is empty")
        item = self.items[self.front]
        self.items[self.front] = None
        self.front = (self.front + 1) % self.capacity
        self.size -= 1
        if self.size <= self.capacity // 4:
            self._resize(self.capacity // 2)
        return item
    def is_empty(self):
        return self.size == 0
    def _resize(self, new_capacity):
        new_items = [None] * new_capacity
        for i in range(self.size):
            new_items[i] = self.items[(self.front + i) % self.capacity]
        self.items = new_items
        self.front = 0
        self.rear = self.size
        self.capacity = new_capacity
    def __str__(self):
        return str([item for item in self.items if item is not None])
fq = FlexiQueue()
fq.enqueue(1)
fq.enqueue(2)
fq.enqueue(3)
print(fq)

fq.dequeue()
print(fq)
fq.enqueue(4)
fq.enqueue(5)
print(fq)
fq.dequeue()
fq.dequeue()
print(fq)
fq.enqueue(6)
fq.enqueue(7)
fq.enqueue(8)
print(fq)
```

```
[1, 2, 3]
[2, 3]
[2, 3, 4, 5]
[4, 5]
[7, 8, 4, 5, 6]
```

```
#Implement Stack using two Queues
class Stack:
    def __init__(self):
        self.q1 = []
        self.q2 = []

    def push(self, item):
        self.q2.append(item)
        while self.q1:
            self.q2.append(self.q1.pop(0))
        self.q1, self.q2 = self.q2, self.q1

    def pop(self):
        if not self.q1:
            raise IndexError("Stack is empty")
        return self.q1.pop(0)

    def peek(self):
        if not self.q1:
            raise IndexError("Stack is empty")
        return self.q1[0]

    def is_empty(self):
        return not self.q1

    def size(self):
        return len(self.q1)

s = Stack()
s.push(1)
s.push(2)
s.push(3)
print(s.size())
print(s.peek())
print(s.pop())
print(s.pop())
s.push(4)
s.push(5)
print(s.size())
print(s.is_empty())
print(s.pop())
print(s.pop())
print(s.pop())
print(s.is_empty())
```

```
3
3
3
2
3
False
5
4
1
True
```

```
#Implement Queue using two Stacks
class Stack:
    def __init__(self):
        self.items = []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        if not self.items:
            raise IndexError("Stack is empty")
        return self.items.pop()
    def is_empty(self):
        return not self.items
class Queue:
    def __init__(self):
        self.in_stack = Stack()
        self.out_stack = Stack()
    def enqueue(self, item):
        self.in_stack.push(item)
    def dequeue(self):
        if self.out_stack.is_empty():
            while not self.in_stack.is_empty():
                self.out_stack.push(self.in_stack.pop())
        return self.out_stack.pop()
    def peek(self):
        if self.out_stack.is_empty():
            while not self.in_stack.is_empty():
                self.out_stack.push(self.in_stack.pop())
        return self.out_stack.items[-1]
    def is_empty(self):
        return self.in_stack.is_empty() and self.out_stack.is_empty()
    def size(self):
        return len(self.in_stack.items) + len(self.out_stack.items)
q = Queue()
q.enqueue(1)
```

```
#Assume that we have Queue with some elements. Write method rotate() which added the existing elements in the reverse order.
from collections import deque

class Queue:
    def __init__(self):
        self.queue = deque()

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.popleft()
        return None

    def is_empty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)

    def rotate(self):
        for _ in range(len(self.queue)):
            self.queue.appendleft(self.queue.pop())

q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
q.rotate()
while not q.is_empty():
    print(q.dequeue())
```

```
1
```