February 21, 2024

**Department of Computer Science and Engineering**
**Islamic University of Technology (IUT)**
**A subsidiary organ of OIC**

Algorithm Engineering Lab 01

**Azmayen Fayek Sabil, 190042122**

# Contents

# 1 Task 1

## 1.1 Implementing Two Stacks in an Array

### 1.1.1 Introduction

In this part, we will implement two stacks in a single array. This is done by dividing the array into two halves, and each stack operates within its designated portion, one from middle to left and another one from middle + 1 to right.

### 1.1.2 Implementation

Below is the Python code for implementing two stacks in an array:

```python
def initialize_two_stacks(size):
    array = [None] * size
    middle = size // 2
    top1 = middle - 1
    top2 = middle
    return array, top1, top2

def push1(array, top1, top2, data):
    if top1 >= 0:
        array[top1] = data
        top1 -= 1
    else:
        print("Stack 1 is Overflow")

def push2(array, top1, top2, data):
    if top2 < len(array):
        array[top2] = data
        top2 += 1
    else:
        print("Stack 2 is Overflow")

def pop1(array, top1):
    if top1 < len(array) - 1:
        top1 += 1
        return array[top1], top1
    else:
        print("Stack 1 is Underflow")
        return None, top1

def pop2(array, top2):
    if top2 > 0:
        top2 -= 1
        return array[top2], top2
    else:
        print("Stack 2 is Underflow")
        return None, top2
```

### 1.1.3 Explanation

- **initialize_two_stacks(size)**: This initializes an array with a given size and sets up the tops of two stacks at the middle of the array.
- **push1(array, top1, top2, data)**: Pushes an element onto Stack 1. Checks for overflow condition.

- **push2(array, top1, top2, data)**: Pushes an element onto Stack 2. Checks for overflow condition.
- **pop1(array, top1)**: Pops an element from Stack 1. Checks for underflow condition.
- **pop2(array, top2)**: Pops an element from Stack 2. Checks for underflow condition.

```
# INPUT
size = 6
array, top1, top2 = initialize_two_stacks(size)
push1(array, top1, top2, 1)
push2(array, top1, top2, 2)
push1(array, top1, top2, 3)
push2(array, top1, top2, 4)

# popping from stack 1
popped_element1, top1 = pop1(array, top1)
print("Popped element from stack 1:", popped_element1)

# popping from stack 2
popped_element2, top2 = pop2(array, top2)
print("Popped element from stack 2:", popped_element2)
```

```
    OUTPUT:
    Popped element from stack 1: 4
    Popped element from stack 2: 3
```

## 1.2 Implementing Stack Using Queues

### 1.2.1 Introduction

In this section, we will implement a stack using two queues. The stack operations, push and pop, will be achieved by utilizing two queues, q1 and q2.

### 1.2.2 Implementation

Below is the Python code for implementing a stack using queues:

```python
from collections import deque

def initialize_stack_using_queues():
    queue1 = deque()
    queue2 = deque()
    return [queue1, queue2]

def push(stack, x):
    # Enqueue x to q2
    stack[1].append(x)

    # Dequeue everything from q1 and enqueue to q2
    while stack[0]:
        stack[1].append(stack[0].popleft())

    # Swap the queues
    stack[0], stack[1] = stack[1], stack[0]

def pop(stack):
    # Dequeue an item from q1 and return it
    if stack[0]:
```

```
            return stack[0].popleft()
    else:
        return None  # Stack is empty
```

### 1.2.3 Explanation

- **initialize_stack_using_queues()**: This function initializes two queues, q1 and q2, and returns them as elements of a stack represented by a list.
- **push(stack, x)**: To push an element onto the stack, the function enqueues the element to q2, then dequeues everything from q1 and enqueues to q2. Finally, it swaps the queues, making q2 the primary queue for the stack.
- **pop(stack)**: To pop an element from the stack, the function dequeues an item from q1. If q1 is empty, it returns None indicating that the stack is empty.

### 1.2.4 Example Usage

Let's see how the stack using queues is utilized:

```
stack_using_queues = initialize_stack_using_queues()
print("Initial stack state:", stack_using_queues)

# Pushing elements onto the stack
push(stack_using_queues, 1)
push(stack_using_queues, 2)
push(stack_using_queues, 3)
print("Stack after pushing elements:", stack_using_queues)

# Popping elements from the stack
popped_element1 = pop(stack_using_queues)
print("Popped element from stack:", popped_element1)

# Pushing more elements onto the stack
push(stack_using_queues, 4)
push(stack_using_queues, 5)

# Popping again
popped_element2 = pop(stack_using_queues)
print("Popped element from stack:", popped_element2)
```

```
    OUTPUT:
    Initial stack state: [deque([]), deque([])]
    Stack after pushing elements: [deque([3, 2, 1]), deque([])]
    Popped element from stack: 3
    Popped element from stack: 5
```

## 1.3 Reverse a Linked List Using a Stack

### 1.3.1 Introduction

In this section, we will implement the reversal of a linked list using a stack. The idea is to push the nodes onto a stack, update the pointers, and pop the nodes back to reverse the order.

### 1.3.2 Implementation

Below is the Python code for reversing a linked list using a stack:

```python
from collections import deque

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

def reverse_linked_list(head):
    if not head or not head.next:
        return head  # Empty or single-node list, no change needed

    stack = deque()

    # Push nodes onto the stack
    current = head
    while current:
        stack.append(current)
        current = current.next

    # Update the head pointer to the last node
    head = stack.pop()
    current = head

    # Pop nodes from the stack and update pointers
    while stack:
        current.next = stack.pop()
        current = current.next

    # Update the next pointer of the last node to NULL
    current.next = None

    return head

def print_linked_list(head):
    current = head
    while current:
        print(current.data, end=" -> ")
        current = current.next
    print("None")
```

### 1.3.3 Explanation

- **Node Class**: This class represents a node in the linked list with data and a next pointer.
- **reverse_linked_list(head)**: This function reverses the linked list using a stack. It first checks if the list is empty or contains a single node. If so, it returns the head unchanged. Otherwise, it pushes nodes onto the stack, updates pointers, and pops nodes back to reverse the order.
- **print_linked_list(head)**: This function prints the linked list for visualization.

### 1.3.4 Example Usage

Let's create and reverse a linked list:

```python
# Creating a linked list: 1 -> 2 -> 3 -> 4 -> 5
head = Node(1)
head.next = Node(2)
head.next.next = Node(3)
head.next.next.next = Node(4)
head.next.next.next.next = Node(5)

print("Original Linked List:")
print_linked_list(head)

# Reversing the linked list using a stack
head = reverse_linked_list(head)

print("Reversed Linked List:")
print_linked_list(head)
```

```
    OUTPUT:
    Original Linked List:
    1 -> 2 -> 3 -> 4 -> 5 -> None
    Reversed Linked List:
    5 -> 4 -> 3 -> 2 -> 1 -> None
```

# 2 Task 2

## 2.1 Deleting the Middle Element from a Stack

### 2.1.1 Introduction

In this section, we will implement a function to delete the middle element from a stack without using any additional data structure. The task involves counting the length of the stack and then removing the middle element.

### 2.1.2 Implementation

Below is the Python code for the deletion of the middle element from a stack:

```
def delete_middle(stack):
    length = 0
    temp_stack = []

    # Counting the length of the stack
    while stack:
        temp_stack.append(stack.pop())
        length += 1

    middle_index = length // 2

    # Deleting the middle element
    for i in range(length):
        if i != middle_index:
            stack.append(temp_stack.pop())
        else:
            temp_stack.pop()
```

### 2.1.3 Explanation

- **delete_middle(stack)**: This function takes a stack as input and deletes its middle element. It first counts the length of the stack by popping elements and storing them in a temporary stack. Then, it identifies the middle index and skips that index while reconstructing the original stack.

### 2.1.4 Example Usage

Let's test the function with some test cases:

```
# Test case 1
stack1 = [34, 3, 31, 40, 98, 92, 23]
delete_middle(stack1)
print("Test Case 1:", stack1)

# Test case 2
stack2 = [3, 5, 1, 4, 2, 8]
delete_middle(stack2)
print("Test Case 2:", stack2)

# Test case 3
stack3 = [3, 4, 2, 8]
delete_middle(stack3)
print("Test Case 3:", stack3)
```

### 2.1.5 Output

```
    OUTPUT:
    Test Case 1: [34, 3, 31, 98, 92, 23]
    Test Case 2: [3, 5, 1, 2, 8]
    Test Case 2: [3, 4, 8]
```

# 3 Task 3

## 3.1 Finding Maximum in Contiguous Subarrays

### 3.1.1 Introduction

In this section, we will implement a solution to find the maximum for every contiguous subarray of size $K$ in an array.

### 3.1.2 Implementation

Below is the Python code for finding the maximum in contiguous subarrays:

```python
# Hardcoded test case
n, k = 9, 3
a = [1, 2, 3, 1, 4, 5, 2, 3, 6]
print("Original array:", a)
print("n and k:", n, k)

ans = []

for i in range(n - k + 1):
    max_val = max(a[i:i+k])
    ans.append(max_val)

print("Ans:")
for value in ans:
    print(value, end=" ")
```

### 3.1.3 Explanation

- **Original array and Parameters:** We start with a hardcoded test case, an array $a$ with values and two integers $n$ and $k$.
- **Loop for Subarrays:** We iterate through the array using a loop to consider every contiguous subarray of size $K$.
- **Maximum in Subarray:** For each subarray, we find the maximum value using the `max()` function.
- **Appending to Result:** The maximum value for each subarray is appended to the result list `ans`.
- **Printing Result:** Finally, we print the result list containing maximum values for each subarray.

### 3.1.4 Output

```
    OUTPUT:
    Original array:  [1, 2, 3, 1, 4, 5, 2, 3, 6]
    n and k: 9 3
    Ans:
    3 3 4 5 5 5 6
    Expected Output for Test Case: 3 3 4 5 5 5 6
```

# 4 Task 4

## 4.1 Intersection of Sorted Linked Lists

### 4.1.1 Introduction

In this section, we will implement a solution to find the intersection of two sorted linked lists. The resulting linked list will be created with its memory, and the original lists will not be changed.

### 4.1.2 Implementation

Below is the Python code for finding the intersection of two sorted linked lists:

```python
class ListNode:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next

def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> ")
        current = current.next
    print("None")

def intersection_linked_lists(list1, list2):
    dummy = ListNode()
    current = dummy

    while list1 and list2:
        if list1.value == list2.value:
            current.next = ListNode(list1.value)
            current = current.next
            list1 = list1.next
            list2 = list2.next
        elif list1.value < list2.value:
            list1 = list1.next
        else:
            list2 = list2.next

    return dummy.next
```

### 4.1.3 Explanation

- **ListNode Class:** This class defines a node in a linked list with a value and a reference to the next node.
- **print_linked_list Function:** This function prints the values of a linked list.
- **intersection_linked_lists Function:** This function takes two sorted linked lists as input and returns a new linked list representing their intersection. It employs a two-pointer approach, where pointers traverse through both lists. The function iterates through both lists, comparing values, and creates a new list with common elements.

  The comparison is done as follows:

- If the value at the current node in *list1* is equal to the value at the current node in *list2*, it implies a common element. The function adds this value to the new linked list and advances both pointers.
- If the value in *list1* is less than the value in *list2*, it means that *list1* may contain a potential common element later. Therefore, the pointer in *list1* is moved to the next node.
- If the value in *list1* is greater than the value in *list2*, it indicates that *list2* may contain a potential common element later. Therefore, the pointer in *list2* is moved to the next node.

This way, the function identifies and constructs a new linked list with elements common to both input lists.

### 4.1.4 Output

```
Test Case 1:
2 -> 4 -> 6 -> None
```

```
Test Case 2:
2 -> 9 -> 12 -> None
```