# Islamic University of Technology

CSE 4810

Algorithm Engineering Lab

---

# Lab 1

---


Tasnimul Hasnat

190041113

CSE 1A


February 21, 2024

# Task 1

## Implement two stacks in an array

```
class TwoStacks:

  def __init__(self, size):
    self.size = size
    self.arr = [None] * size
    self.top1 = -1
    self.top2 = size

  def push1(self, item):
    if self.top1 < self.top2 - 1:
      self.top1 += 1
      self.arr[self.top1] = item
    else:
      print("Stack Overflow")

  def push2(self, item):
    if self.top1 < self.top2 - 1:
      self.top2 -= 1
      self.arr[self.top2] = item
    else:
      print("Stack Overflow")

  def pop1(self):
    if self.top1 >= 0:
      item = self.arr[self.top1]
      self.top1 -= 1
      return item
    else:
      print("Stack 1 is Empty")

  def pop2(self):
    if self.top2 < self.size:
      item = self.arr[self.top2]
      self.top2 += 1
      return item
    else:
      print("Stack 2 is Empty")

  def print_stack1(self):
    print("Stack 1: ", end="")
    for i in range(self.top1, -1, -1):
      print(self.arr[i], end=" ")
    print()

  def print_stack2(self):
    print("Stack 2: ", end="")
    for i in range(self.top2, self.size):
      print(self.arr[i], end=" ")
    print()
```

### Driver Code

```
stacks = TwoStacks(6)
stacks.push1("p")
stacks.push1("e")
stacks.push1("w")
stacks.push2("p")
stacks.push2("e")
stacks.push2("w")
stacks.print_stack1()
stacks.print_stack2()
print(stacks.pop1())
print(stacks.pop2())
stacks.print_stack1()
stacks.print_stack2()
```

### Output

```
Stack 1: w e p
Stack 2: w e p
w
w
Stack 1: e p
Stack 2: e p
```

The `TwoStacks` class implements two stacks using a single array. The class has methods to push elements onto each stack (`push1` and `push2`), pop elements from each stack (`pop1` and `pop2`), and print the elements of each stack (`print_stack1` and `print_stack2`).

The class maintains two pointers `top1` and `top2`, which represent the top elements of each stack. Initially, `top1` is set to -1 and `top2` is set to the size of the array. When pushing elements onto a stack, the corresponding pointer is incremented or decremented accordingly, and the item is added to the array. When popping elements, the top item is retrieved, and the corresponding pointer is adjusted.

It also ensures that the stacks do not overflow by checking if there is space available before pushing elements (if `self.top1 < self.top2 - 1`). Similarly, it handles underflow conditions by checking if there are elements present before popping (if `self.top1 >= 0` for stack 1 and if `self.top2 < self.size` for stack 2).

**Time Complexity:**

- **Push Operation:** $O(1)$
- **Pop Operation:** $O(1)$

**Space Complexity:** $O(n)$

## Implement stack using queues

```
class Stack:

  def __init__(self):
    self.queue1 = []
    self.queue2 = []

  def push(self, item):
    self.queue1.append(item)

  def pop(self):
    if not self.queue1:
      return None

    while len(self.queue1) > 1:
      self.queue2.append(self.queue1.pop(0))

    popped_item = self.queue1.pop(0)

    self.queue1, self.queue2 = self.queue2, self.queue1

    return popped_item

  def top(self):
    if not self.queue1:
      return None

    while len(self.queue1) > 1:
      self.queue2.append(self.queue1.pop(0))

    top_item = self.queue1[0]

    self.queue2.append(self.queue1.pop(0))
    self.queue1, self.queue2 = self.queue2, self.queue1

    return top_item

  def empty(self):
    return len(self.queue1) == 0
```

### Driver Code

```
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
print("Top ->",stack.top())
stack.pop(); print("Popping")
print("Now the Top ->",stack.top())
print("Is stack empty? ",stack.empty())
```

**Output**

```
Top -> 3
Popping
Now the Top -> 2
Is stack empty?  False
```

The class `Stack` implements a stack using two queues. The class initializes two empty queues (`queue1` and `queue2`). The `push` method adds an item to `queue1`, simulating a push operation onto the stack. The `pop` method removes and returns the top item from the stack. It achieves this by transferring all elements except the last one from `queue1` to `queue2`, then popping the last element from `queue1` and swapping the queues. The `top` method returns the top item from the stack without removing it, similar to `pop` but retains the last element in `queue1`. The `empty` method checks if the stack is empty by verifying if `queue1` is empty. This implementation offers a stack interface using queues, utilizing *FIFO (First In, First Out)* behavior of queues to simulate *LIFO (Last In, First Out)* behavior of stacks.

**Time Complexity:**

- **Push Operation:** $O(n)$
- **Pop Operation:** $O(1)$

**Space Complexity:** $O(n)$

## Reverse a link list using stack

```python
class Node:

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


class LinkedList:

  def __init__(self):
    self.head = None

  def push(self, data):
    new_node = Node(data)
    new_node.next = self.head
    self.head = new_node

  def reverse(self):
    if self.head is None:
      return

    stack = []
    current = self.head

    while current is not None:
      stack.append(current)
      current = current.next

    self.head = stack.pop()
    current = self.head
    while stack:
      current.next = stack.pop()
      current = current.next
    current.next = None

  def print(self):
    current = self.head
    while current:
      print(current.data, end=" ")
      current = current.next
    print()
```

**Driver Code**

```
ll = LinkedList()
ll.push(1)
ll.push(2)
ll.push(3)
ll.push(4)
ll.push(5)

print("Original Linked List:")
ll.print()

ll.reverse()

print("Reversed Linked List:")
ll.print()
```

**Output**

```
Original Linked List:
5 4 3 2 1
Reversed Linked List:
1 2 3 4 5
```

The `reverse` method in the `LinkedList` class is designed to reverse the order of nodes within the linked list. It begins by checking if the list is empty, returning if so. Next, it initializes an empty stack and traverses the list, pushing each node onto the stack. Once all nodes are on the stack, the method updates the `head` pointer to the last node, effectively making it the new head of the list. Then, it iterates through the stack, popping nodes one by one and updating their next pointers to point to the previously popped node. This process effectively reverses the order of nodes in the list. Finally, the next pointer of the last node is set to None, marking the end of the reversed list.

This approach efficiently reverses the linked list by leveraging a stack to reorder the nodes in place.

**Time Complexity:** $O(n)$
**Space Complexity:** $O(n)$

## Task 2

Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in the array. Elements for which no greater element exist, consider the next greater element as -1.

```
def NGE(arr):
    stack =[]
    stack.append(arr[0])

    top = -1
    next = -1

    for i in range(1,len(arr)):
        next = arr[i]
        if len(stack)!= 0:
            top= stack.pop()

            if top > next :
                stack.append(top)
            while top<next:
                print(top,'->', next)
                if len(stack) == 0:
                    break
                top = stack.pop()

        stack.append(next)

    while( len(stack)!=0 ):
        top= stack.pop()
        next =-1
        print(top, "->",next)
```

### Driver Code

```
print("Test case 1")
arr = [4,5,2,25]
print("Input:", arr, "\nOutput:")
NGE(arr)
print("\nTest case 2")
arr2 = [13,7,6,12]
print("Input:", arr2, "\nOutput:")
NGE(arr2)
```

**Output**

```
Test case 1
Input: [4, 5, 2, 25]
Output:
4 -> 5
2 -> 25
5 -> 25
25 -> -1

Test case 2
Input: [13, 7, 6, 12]
Output:
6 -> 12
7 -> 12
12 -> -1
```

The function `NGE()`, is designed to find the **Next Greater Element (NGE)** for each element in the input array arr. It utilizes a stack data structure to efficiently determine the **NGE** for each element. Here's a breakdown of how it works:

- It initializes an empty stack.
- It iterates through each element in the input array.
- For each element, it compares it with the top element of the stack.
- If the current element is greater than the top element of the stack, it prints the NGE for the top element (which is the current element), and repeats this process until the current element is not greater than the top element of the stack.
- It then pushes the current element onto the stack.
- Finally, it prints any remaining elements in the stack with a Next Greater Element of -1.

This implementation of the algorithm efficiently finds the Next Greater Element for each element in the input array using a stack.

**Time Complexity:** $O(n)$
**Space Complexity:** $O(n)$

## Task 3

Given a number N, write a function that generates and prints all binary numbers with decimal values from 1 to N.

```
from queue import Queue
def generatePrintBinary(n):
  q = Queue()

  q.put("1")

  while (n > 0):
    n -= 1
    s1 = q.get()
    print(s1)

    s2 = s1
    q.put(s1 + "0")
    q.put(s2 + "1")
```

### Driver Code

```
print("Test Case 1 for n = 2")
generatePrintBinary(2)

print("Test Case 1 for n = 11")
generatePrintBinary(11)
```

### Output

```
Test Case 1 for n = 2
1
10
Test Case 1 for n = 5
1
10
11
100
101
```

The `generatePrintBinary()` function, utilizes a *queue* data structure to generate and print binary numbers up to a specified limit **n**. Initially, it imports the `Queue` class from the queue module. It then initializes a queue `q` and adds the string "1" to it. The function enters a while loop, decrementing n with each iteration. Within the loop, it retrieves and prints the first element s1 from the queue. It then creates two new strings, s1 + "0" and s2 + "1", by appending "0" and "1" to s1 respectively, and enqueues them into the queue. This process continues until n reaches 0, printing binary numbers in ascending order.

**Time Complexity:** $O(n)$
**Space Complexity:** $O(n)$

## Task 4

Given two lists sorted in increasing order, create and return a new list representing the intersection of the two lists. The new list should be made with its memory — the original lists should not be changed.

```
def intersection(list1, list2):
  result = LinkedList()
  curr1 = list1.head
  curr2 = list2.head

  while curr1 is not None and curr2 is not None:
    if curr1.data == curr2.data:
      result.push(curr1.data)
      curr1 = curr1.next
      curr2 = curr2.next
    elif curr1.data < curr2.data:
      curr1 = curr1.next
    else:
      curr2 = curr2.next

  return result
```

### Driver Code

```
list1 = LinkedList()
list1.push(14)
list1.push(12)
list1.push(10)
list1.push(9)
list1.push(8)
list1.push(2)
list1.push(1)
print("First List: ",end="")
list1.print()

list2 = LinkedList()
list2.push(13)
list2.push(12)
list2.push(11)
list2.push(9)
list2.push(4)
list2.push(3)
list2.push(2)
print("\nSecond List: ",end="")
list2.print()

print("\nIntersected List: ",end="")
result = intersection(list1, list2)
result.reverse()
result.print()
```

**Output**

```
First List: 1 2 8 9 10 12 14

Second List: 2 3 4 9 11 12 13

Intersected List: 2 9 12
```

The `intersection()` function takes two sorted linked lists (`list1` and `list2`) as input and returns a new linked list containing the elements that are common to both input lists.

It initializes an empty linked list called `result` to store the intersection elements. Then, it sets two pointers, curr1 and curr2, to the heads of list1 and list2 respectively.

The function iterates through both lists simultaneously using a while loop, comparing the data of the current nodes pointed to by curr1 and curr2. If the data in both nodes is equal, it means there's an intersection, so it pushes that data onto the result list and moves both pointers to their respective next nodes. If the data in curr1 is less than that in curr2, it moves curr1 to its next node, and vice versa if curr2′s data is less.

This process continues until either of the pointers reaches the end of its list (i.e., becomes None). At this point, the function has completed finding the intersection, and it returns the result linked list containing the common elements.

**Time Complexity:** $O(m+n)$
**Space Complexity:** $O(\max(m,n))$
*where m and n are the size of the first and second list respectively.*