Assignment 4

Question 1

The stack implementation using 2 queues can be done as follows:

- Let the two queues be $q1$ and $q2$.
- **PUSH operation:**
    - Simply enqueue the element to be pushed into queue $q1$.
- **POP operation:**
    - Since we need access to the most recently pushed element (as in a stack), we dequeue elements one-by-one from the head of the queue $q1$, and enqueue them into the other queue $q2$ until we reach the last element in $q1$.
    - When $q1$ has no more elements left, this last element which we have dequeued is the popped element that we need to return, and it is *NOT* enqueued into $q2$ as the POP operation **removes** the last-in element.
    - $q1$ and $q2$ are interchanged, so that the subsequent PUSH, POP, and TOP operations occur on the queue which has the elements (since all the elements have been moved into $q2$).
    - Therefore, the older $q2$ is henceforth $q1$, and the older $q1$ is henceforth $q2$.
- **TOP operation:**
    - Exactly the same as the POP operation, except that the last element which is dequeued from $q1$ is returned *AND* enqueued into $q2$ without being discarded.
    - This is because the TOP operation **does not remove** the last-in element of the stack.

Let us demonstrate this algorithm with an example sequence of operations for easier understanding.

1. **PUSH 4:** Let us first PUSH **4** into the stack as above (goes into $q1$).

4	-	-	-	||	-	-	-	-
q1				-	**q2**			

2. **PUSH 5:** Next we PUSH **5** into the stack (enqueues into $q1$).

5	4	-	-	||	-	-	-	-
q1				-	**q2**			

3. **PUSH 6:** Next we PUSH **6** into the stack (enqueues into $q1$).

6	5	4	-	||	-	-	-	-
q1				-	**q2**			

4. **TOP:** The steps described above for the TOP operation are taken (all elements enqueued one-by-one into $q2$ and last element returned, and then the queues are interchanged).

```
6   5   -   -    ||    4   -   -   -
```
q1 - **q2**

```
6   -   -   -    ||    5   4   -   -
```
q1 - **q2**

```
-   -   -   -    ||    6   5   4   -
```
q1 - **q2**

6 is returned as the top element and the queues are interchanged:

```
-   -   -   -    ||    6   5   4   -
```
q2 - **q1**

5. **TOP:** Once again, the TOP operation is performed to demonstrate that the TOP operation does not pop/remove an element (same top element as before is expected).

```
4   -   -   -    ||    6   5   -   -
```
q2 - **q1**

```
5   4   -   -    ||    6   -   -   -
```
q2 - **q1**

```
6   5   4   -    ||    -   -   -   -
```
q2 - **q1**

6 is returned as the top element and the queues are interchanged:

```
6   5   4   -    ||    -   -   -   -
```
q1 - **q2**

6. **POP:** POP operation is done as earlier described (all elements dequeued from $q1$ and enqueued into $q2$ until last element, last element popped without enqueueing into $q2$, and queues interchanged).

```
6   5   -   -    ||    4   -   -   -
```
q1 - **q2**

```
6   -   -   -    ||    5   4   -   -
```
q1 - **q2**

```
-   -   -   -    ||    5   4   -   -
```
q1 - **q2**

```
6
```

6 is popped and the queues are interchanged:

```
-  -  -  -   ||   5  4  -  -
q2           -  q1
```

7. **TOP:** Now, element 5 is returned as the top element.

```
4  -  -  -   ||   5  -  -  -
q2           -  q1
```

```
5  4  -  -   ||   -  -  -  -
q2           -  q1
```

5 is returned as the top element and the queues are interchanged:

```
5  4  -  -   ||   -  -  -  -
q1           -  q2
```

8. **POP:** This time, element **5** is popped.

```
5  -  -  -   ||   4  -  -  -
q1           -  q2
```

```
-  -  -  -   ||   4  -  -  -
q1           -  q2
5
```

5 is popped and the queues are interchanged:

```
-  -  -  -   ||   4  -  -  -
q2           -  q1
```

9. **PUSH 8:** **8** is pushed into the stack (goes into the current $q1$ - not necessarily the original $q1$).

```
-  -  -  -   ||   8  4  -  -
q2           -  q1
```

10. **TOP:** Top element is returned as **8**.

```
4  -  -  -   ||   8  -  -  -
q2           -  q1
```

```
8  4  -  -   ||   -  -  -  -
q2           -  q1
```

8 is returned as the top element and the queues are interchanged:

```
8  4  -  -    ||    -  -  -  -
```

q1 - **q2**

11. **POP: 8** is popped.

```
8  -  -  -    ||    4  -  -  -
```

q1 - **q2**

```
-  -  -  -    ||    4  -  -  -
```

q1 - **q2**

8

8 is popped and the queues are interchanged:

```
-  -  -  -    ||    4  -  -  -
```

q2 - **q1**

12. **POP: 4** is popped.

```
-  -  -  -    ||    -  -  -  -
```

q2 - **q1**

4

4 is popped and the queues are interchanged:

```
-  -  -  -    ||    -  -  -  -
```

q1 - **q2**

13. **POP:** Since the stack is empty, the user is alerted that the stack is empty and no element is returned.
14. **POP:** Once again, since the stack is empty, the user is alerted that the stack is empty and no element is returned.

Below is a code implementation of this model of the two-queue stack, and its functioning demonstrated using the example discussed above.

```
# Importing Required Packages

from queue import Queue
from queue import Empty
```

```
# Writing a class for our implementation of a stack

```

```python
class Stack:
    def __init__(self):
        self.q1 = Queue()
        self.q2 = Queue()
        # Creating the queue pair of q1, q2
        self.queue_pair = [self.q1, self.q2]

    # Function to reverse the current order of queues for POP and TOP
    def switch(self):
        self.queue_pair = list(reversed(self.queue_pair))

    # Function for the PUSH operation
    def push(self, ele):
        self.queue_pair[0].put(ele)
        # Displaying a message whenever an element is pushed
        # print("PUSH: Element pushed = ", ele)

    # Function for the POP operation
    def pop(self):
        while True:
            try:
                # Trying to remove and get the "head" of the current q1
                # current q1 is the queue at index 0 of the queue_pair
                elm = self.queue_pair[0].get(timeout = 0.1)

            except Empty:
                # Returning "STACK IS EMPTY" if the current q1 is empty
                return "STACK IS EMPTY!"

            # If the "head" of q1 was the last element of q1, break from loop
            if self.queue_pair[0].empty():
                break
            # If the "head" was NOT the last element of q1,
            # only then enqueue into q2 (last element should be popped)
            else:
                self.queue_pair[1].put(elm)
        # Reverse order of the queues
        self.switch()
        # Return the popped element
        return elm

    # Function for the TOP operation
    def top(self):
        while True:
            try:
                # Trying to remove and get the "head" of the current q1
                # current q1 is the queue at index 0 of the queue_pair
                elm = self.queue_pair[0].get(timeout = 0.1)

            except Empty:
                # Returning "STACK IS EMPTY" if the current q1 is empty
                return "STACK IS EMPTY!"

            # ALWAYS enqueue into q2 (last element should not be popped)
            self.queue_pair[1].put(elm)
            # If the "head" of q1 was the last element of q1, break from loop
            if self.queue_pair[0].empty():
                break
                break
        # Reverse order of the queues
        self.switch()
        # Return the top element of the stack
        return elm

```

```
# DEMONSTRATION OF THE STACK IMPLEMENTATION USING SOME OPERATIONS

stack = Stack()

stack.push(4)           # PUSH 4
stack.push(5)           # PUSH 5
stack.push(6)           # PUSH 6

print()    # Empty prints to print empty lines and make the output easy to read
print("TOP: Top element (peek) =", stack.top())  # TOP: Should show top element 6
print("TOP: Top element (peek) =", stack.top())  # TOP: Should still show top
                                                 # element (6) without popping
print()
print("POP: Popped element =", stack.pop())      # POP: Should pop 6
print("TOP: Top element (peek) =", stack.top())  # TOP: Top element must now be 5
print("POP: Popped element =", stack.pop())      # POP: Should pop 5
print()

stack.push(8)           # PUSH 8

print()
print("TOP: Top element (peek) =", stack.top())  # TOP: Top element must now be 8
print()
print("POP: Popped element =", stack.pop())      # POP: Should pop 8
print("POP: Popped element =", stack.pop())      # POP: Should pop 4
print()
print("POP:", stack.pop())                      # POP: Should say 'STACK IS EMPTY!'
print("POP:", stack.pop())                      # POP: Should say 'STACK IS EMPTY!'
```

```
TOP: Top element (peek) = 6
TOP: Top element (peek) = 6

POP: Popped element = 6
TOP: Top element (peek) = 5
POP: Popped element = 5


TOP: Top element (peek) = 8

POP: Popped element = 8
POP: Popped element = 4

POP: STACK IS EMPTY!
POP: STACK IS EMPTY!
```

Question 2

Given:

- Keys $5, 28, 19, 15, 20, 33, 12, 17, 10$

Conditions:

1. Hash function $h(k) = k \mod 9$
2. Hash table has 9 slots.
3. Collisions are resolved by chaining.

Let us insert the keys one-by-one into the hash table to demonstrate what happens.

1. Let us first insert the key 5.
   Hash value $= h(5) = 5 \mod 9 = 5$.
   So we put the key 5 in the corresponding hash table location 5.

0	1	2	3	4	5	6	7	8
-	-	-	-	-	5	-	-	-

2. Next we insert the key 28.
   Hash value $= h(28) = 28 \mod 9 = 1$.
   So we put the key 28 in the corresponding hash table location 1.

0	1	2	3	4	5	6	7	8
-	28	-	-	-	5	-	-	-

3. Next we insert the key 19.
   Hash value $= h(19) = 19 \mod 9 = 1$ and this is a **collision**.
   This is where chaining comes into play. The entry at the hash table location will become a linked list/chain of elements containing a record of all elements with the same hash value. So we put the key 19 in the corresponding hash table location 1, chained to the already existing entry 28 in the form of a linked list.

   (The character '|' has been used to represent a downward arrow, as it is hard to draw a downward arrow on a computer-generated report.)

0	1	2	3	4	5	6	7	8
-	28	-	-	-	5	-	-	-
	|							
	19							

4. Next we insert the key 15.
   Hash value $= h(15) = 15 \mod 9 = 6$.
   So we put the key 19 in the corresponding hash table location 6.

0	1	2	3	4	5	6	7	8
-	28	-	-	-	5	15	-	-
	|							
	19							

5. Next we insert the key 20.
   Hash value $= h(20) = 20 \mod 9 = 2$.
   So we put the key 20 in the corresponding hash table location 2.

0	1	2	3	4	5	6	7	8
-	28	20	-	-	5	15	-	-
	|							
	19							

6. Next we insert the key 33.
   Hash value $= h(33) = 33 \mod 9 = 6$ and there is a **collision**.
   So we put the key 33 in the corresponding hash table location 6, chained to the key 15 as a linked list.

0	1	2	3	4	5	6	7	8
-	28	20	-	-	5	15	-	-
	|					|		
	19					33		

7. Next we insert the key 12.
   Hash value $= h(12) = 12 \mod 9 = 3$.
   So we put the key 12 in the corresponding hash table location 3.

0	1	2	3	4	5	6	7	8
-	28	20	12	-	5	15	-	-
	|					|		
	19					33		

8. Next we insert the key 17.
   Hash value $= h(17) = 17 \mod 9 = 8$.
   So we put the key 17 in the corresponding hash table location 8.

0	1	2	3	4	5	6	7	8
-	28	20	12	-	5	15	-	17
	|					|		
	19					33		

9. Finally, we insert the key 10.
   Hash value $= h(10) = 10 \mod 9 = 1$, which is a **collision**.
   So we put the key 10 in the corresponding hash table location 1, chained to the last key with hash value 1, which was 19.

0	1	2	3	4	5	6	7	8
-	28	20	12	-	5	15	-	17
	|					|		
	19					33		
	|							
	10							

This is the final state of the hash table chained with linked lists.

Question 3

Given:

- A binary search tree T whose keys are distinct.

Conditions:

1. Right subtree of node x is empty.
2. y is the successor of $x \implies x < y$ (since keys are distinct) and any other node z in T must satisfy $z < x$ or $z > y$. In other words, no z can lie in between x and y.

Assumption for the entire problem: A node is an ancestor of itself.

Step 1: **Let us first show that y must be an ancestor of x.**

- Let us initially assume that y is **not** an ancestor of x. Since y is a successor of x, $x < y$ must hold.
- This is possible if y is a right child of x, but that cannot be the case here, since it is given that the right subtree of node x is empty.
- Therefore, the only possibility is if there exists some common ancestor z for both x and y, such that x is in the left subtree and y is in the right subtree of z.
- However, this would mean that $x < z < y$ according to the property of a binary search tree. This must not be the case as, according to condition 2, the successor of x must be y.
- Thus our initial assumption is wrong, and y **must be an ancestor of** x by contradiction.

Step 2: **Next, let us show that $y.left$ must also be an ancestor of x.**

- According to the assumed definition of a node, a node is an ancestor of itself.

- Let us initially assume that $y.left$ is **not** an ancestor of x. Since y must be an ancestor of x, the only other way that can be possible is if $y.right$ is an ancestor of x.
- However, if $y.right$ is an ancestor of x, by the binary search tree property, $x > y$ which violates condition 2.
- Thus our initial assumption is wrong, and $y.left$ **must be an ancestor of** x by contradiction.

***Step 3:* Finally, let us show that y is the lowest such ancestor of x whose left child is also an ancestor of x.**

- Let us initially assume that y is **not** the lowest such ancestor.
- Then, there exists a node z which **is** the lowest such ancestor, whose left child is also an ancestor of x.
- Since $y.left$ is an ancestor of x as shown in Step 2, x belongs to the left subtree of y.
- Since z is also an ancestor of x and is lower in the tree than y, z must also belong to the left subtree of y. Therefore $z < y$.
- Also, $z.left$ is an ancestor of x according to our initial assumption $\implies$ x is in the left subtree of $z \implies x < z$.
- Therefore, we have $x < z < y$, with our initial assumption. However, this violates condition 2.
- Thus our initial assumption is wrong, and y **must be the lowest ancestor of x, whose left child is also an ancestor of x,** by contradiction.

Question 4

The Algorithm

- The explicit stack will be used to store all the intermediate results while generating all the permutations of the list $[1, 2, 3, \ldots, N]$.
- In this stack, each element will be in the form of a tuple which contains 2 sublists of the list $[1, 2, 3, \ldots, N]$.
- The first sublist will be used to generate the permutations.
- The second sublist will contain the numbers that are yet to be added to the permutation being generated in the first sublist.
- The permutations will be generated in a Depth-First fashion as follows:
  1. We begin with the first stack element as $([], [1, 2, 3, \ldots, N])$. It is PUSHed into the stack.
  2. We generate N daughter elements from this single parent element, by taking each number in the second sublist as the singleton member of the first sublist, in each element. This is done as follows:

$$\begin{aligned}([], [1, 2, 3, \ldots, N]) \implies & ([1], [2, 3, 4, \ldots, N]), \\ & ([2], [1, 3, 4, \ldots, N]), \\ & ([3], [1, 2, 4, \ldots, N]), \\ & \ldots, ([N], [1, 2, 3, 4, \ldots, N-1])\end{aligned}$$

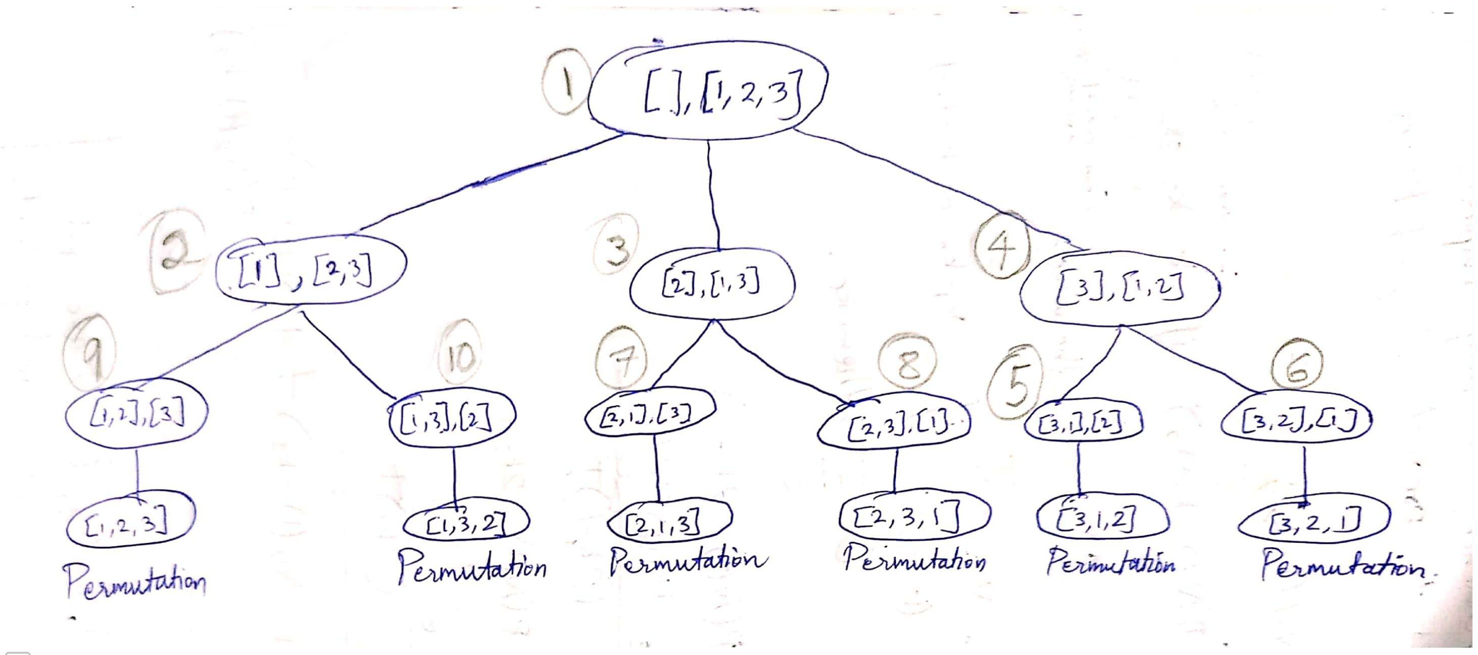
  3. All these N elements are PUSHed into the stack.

4. We now POP the topmost element of the stack. Without loss of generality, let us assume that $([1],[2,3,\ldots,N])$ is the popped element.
5. We repeat 2., except we generate only $N-1$ daughter elements from this single parent element (since the second sublist has only $N-1$ numbers to include in the permutation being generated in the first sublist). This is done as follows:

$$\begin{aligned}([1],[2,3,4,\ldots,N]) \implies & ([1,2],[3,4,\ldots,N]),\\ & ([1,3],[2,4,\ldots,N]),\\ & ([3,4],[1,2,\ldots,N]),\\ & \ldots,([1,N],[2,3,4,\ldots,N-1])\end{aligned}$$

6. All these $N-1$ elements are PUSHed into the stack.
7. When we reach a point where the second sublist has only one number left, e.g., $([1,4,10,\ldots,N-1],[N])$, then the two lists are appended to generate a full permutation of the original list, e.g., $([1,4,10,\ldots,N-1,N])$, and this element is POPped from the stack and returned as one of the required permutations.
8. We repeat this process until the stack is empty, i.e., all the elements are POPped and have been used for generating permutations.

- An example with $N=3$ has been used to demonstrate through a simple diagram on paper.



- The numbers in pencil next to each element indicate one of the possible orders in which the elements would be generated and added to the stack.
- It is noteworthy that an element is added only when its "parent" element has been already popped. For example, the element $([1],[2,3])$ labelled "2" is PUSHed into the stack only after the the parent element $([],[1,2,3])$ has been POPped and used to generate the daughter elements labelled "2", "3", and "4".
- Similarly, the element "5" which gives rise to a permutation is generated only after the parent element "4" has been POPped and used to generate "5" and "6".

- The lowest layer of daughter elements (those with only one number in the second sublist) need not be PUSHed into the stack. For example, these elements in the above diagram are labelled "5", "6", "7", "8", "9", and "10".
  When such an element is detected, the sublists are combined and returned directly as a permutation to the output.

A code implementation of this algorithm is presented below:

```python
# Using the Stack class created in Question 1 for the explicit stack
# Please make sure to run the cell containing the Stack class in Question 1

def perms_generator(N):

    # N - Input: the number N in the list [1,2,...,N] for which all
    # possible permutations are to be generated

    exp_stack = Stack() # Explicit stack created

    # The first element, i.e., ([], [1, 2,.., N]) is PUSHed
    exp_stack.push(([], list(range(1,N+1))))

    # perms - Output : List for storing and returning all the permutations
    perms = []

    # While loop that runs until the stack is empty
    while not exp_stack.queue_pair[0].empty():
        # Popping the topmost element
        elm = exp_stack.pop()

        # If the length of second sublist is 1, generate permutation directly
        if len(elm[1]) == 1:
            perms.append(elm[0] + elm[1])

        # Else, generate all the daughter elements using a for loop
        else:
            for num in elm[1]:
                second_sub = elm[1].copy()
                second_sub.remove(num)
                first_sub = elm[0].copy()
                first_sub.append(num)
                # PUSH all daughter elements into the explicit stack
                exp_stack.push([first_sub, second_sub])

    # Returning the list containing all permutations of [1,2,3,...,N]
    return perms
```

```python
# DEMONSTRATION OF THE ABOVE FUNCTION USING [1,2,3,4,5]

perms_of_12345 = perms_generator(5)
print("Permutations of [1,2,3,4,5] are", perms_of_12345)
```

```
Permutations of [1,2,3,4,5] are [[5, 4, 3, 2, 1], [5, 4, 3, 1, 2], [5, 4, 2, 3, 1],
```

Question 5

Given:

- An n-node binary search tree.

Conditions:

1. Only tree rotations must be used to change the tree.

Assumption made for the problem: The given binary tree is a binary search tree.

This assumption needs to be made as tree rotations are order-preserving operations, and we can only go from one BST to another BST, both of which correspond to the **same ordered set** of nodes.
(We cannot change the order of in-order traversal of the tree using just rotations, so it is not possible to go from any arbitrary BT to another, as given in the question in the assignment.)

***Step 1:* Let us first show that any arbitrary binary tree can be converted into a right chain in at most n-1 right rotations.**

- Consider an arbitrary n-node binary tree. Let the root and all successive right children form the initial right-going chain.
- Starting from the rightmost child, going up the right chain, upto the root, do the following steps:
    1. If the current chain node c has a left child, perform a right rotation on this parent chain node c.
       This process brings the left child of c into the chain, and does not remove any existing chain nodes from the chain. **Therefore, each such right rotation brings 1 more node of the tree into the right-going chain.**
    2. If the above right rotation results in the current node c getting another different left child, repeat 1. Repeat until the current node c does not have any left child/left subtree.
    3. Once the current node c has no left child, move the "current-node pointer" up the right-going chain by one level. Repeat 1,2,3 with the **new** current node. Keep going until we have obtained the full right-going chain.
- Now, each right rotation brings 1 node into the chain as seen earlier. In an n-node binary tree, the initial chain must have *at least* 1 node - the root itself, even if it does not have any right descendants. Therefore, in a worst-case situation where only the root forms the initial right-going chain, we will need to bring n-1 nodes into the chain using right rotations.
- **Thus, we will need *at most* n-1 right rotations to converted any n-node binary tree into a right-going chain.**

***Step 2:* Let us show that $O(n)$ rotations are required to convert any n-node binary search tree into any other n-node binary search tree.**

- Since the proof in *Step 1* holds for any binary tree, it must hold for a binary search tree as well.

- Let us denote our initial BST as T_1 and our final BST as T_2, both of which correspond to the same ordered sequence (hence, the right-going chain of both trees will be the same). Let us denote the common right-going chain with R.
- Say we need k right rotations, denoted by $r_1, r_2, r_3, \ldots, r_k$ to convert T_1 into the right-going chain R.
- And we need l right rotations, denoted by $s_1, s_2, s_3, \ldots, s_l$ to convert T_2 into the right-going chain R.
- Since a left rotation is the inverse operation of a right-rotation, we can convert R back to T_2, using a sequence of l left-rotations. Let us denote these by $s'_1, s'_2, s'_3, \ldots, s'_l$
- Therefore, the sequence of rotations $r_1, r_2, \ldots, r_k, s'_1, s'_2, \ldots, s'_l$ takes us from any T_1 to any T_2. The number of rotations $= k + l$.
- We have, from *Step 1*:
$$\begin{aligned} & k \leq n - 1 \\ \implies & k < n \end{aligned}$$
and
$$\begin{aligned} & l \leq n - 1 \\ \implies & l < n \end{aligned}$$
- Therefore,
$$\begin{aligned} & k + l < 2n \\ \implies & k + l < cn \end{aligned}$$
for all $n \geq 1$, where $c = 2$.
- **Hence, the number of rotations is $O(n)$.**