Assignment- 6

Anshika

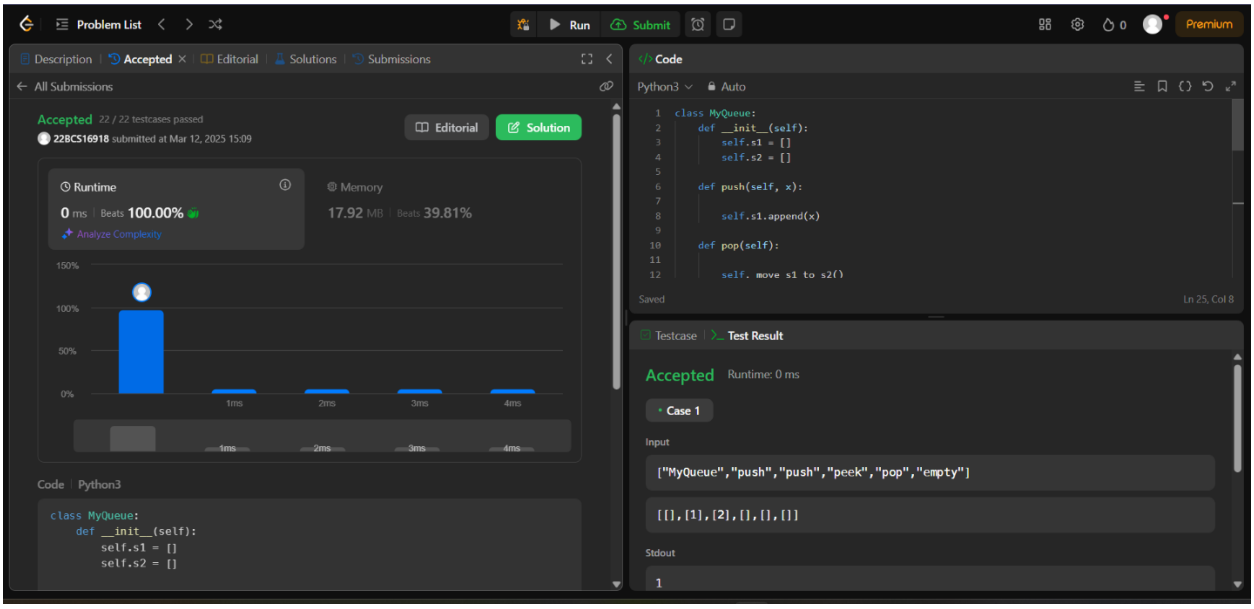
22BCS16918

611 - ‘B’

1. Implement queue using stack

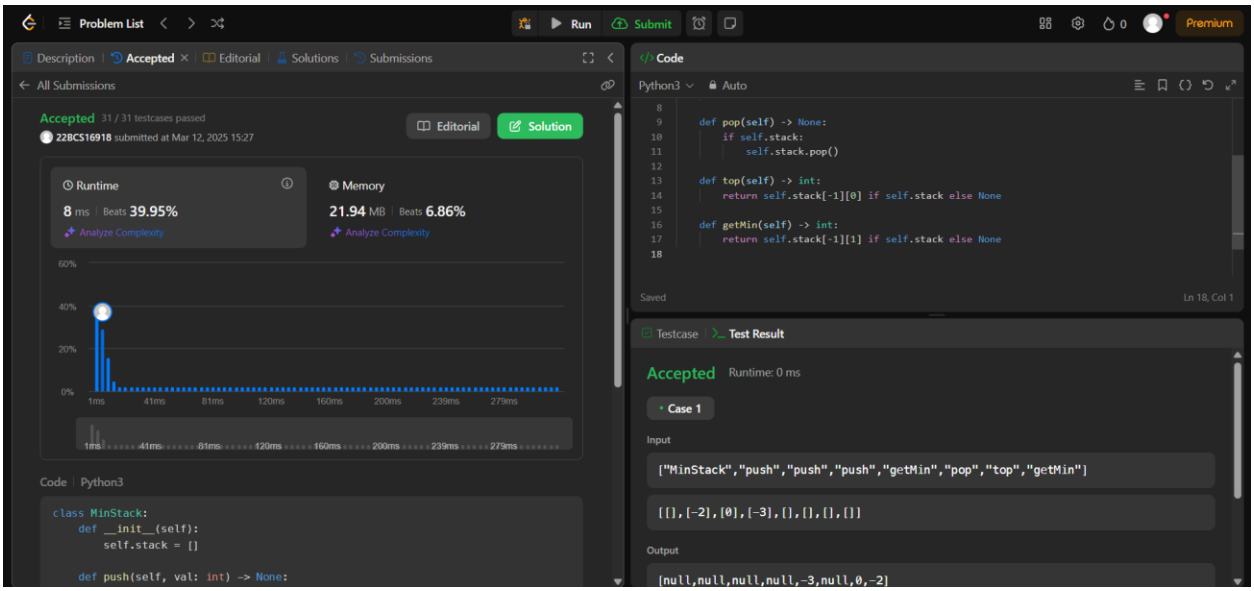
```
class MyQueue:
    def __init__(self):
        self.s1 = []
        self.s2 = []

    def push(self, x):

        self.s1.append(x)

    def pop(self):

        self._move_s1_to_s2()
        return self.s2.pop()

    def peek(self):

        self._move_s1_to_s2()
        return self.s2[-1]

    def empty(self):

        return not self.s1 and not self.s2

    def _move_s1_to_s2(self):

        if not self.s2:
            while self.s1:
                self.s2.append(self.s1.pop())

# Example
queue = MyQueue()
```

```
queue.push(1)
queue.push(2)
print(queue.peek())
print(queue.pop())
print(queue.empty())
```



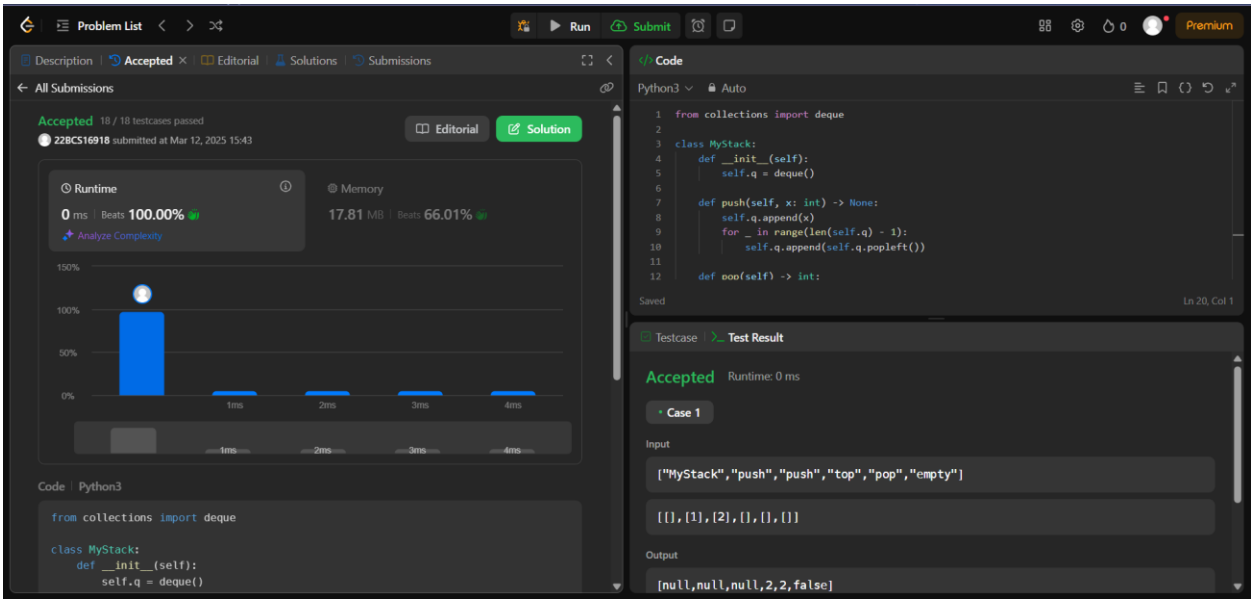
2. Min Stack using Two stacks

```
class MinStack:
    def __init__(self):
        self.stack = []

    def push(self, val: int) -> None:
        min_val = min(val, self.stack[-1][1] if self.stack else val)
        self.stack.append((val, min_val))

    def pop(self) -> None:
        if self.stack:
            self.stack.pop()

    def top(self) -> int:
        return self.stack[-1][0] if self.stack else None

    def getMin(self) -> int:
        return self.stack[-1][1] if self.stack else None
```



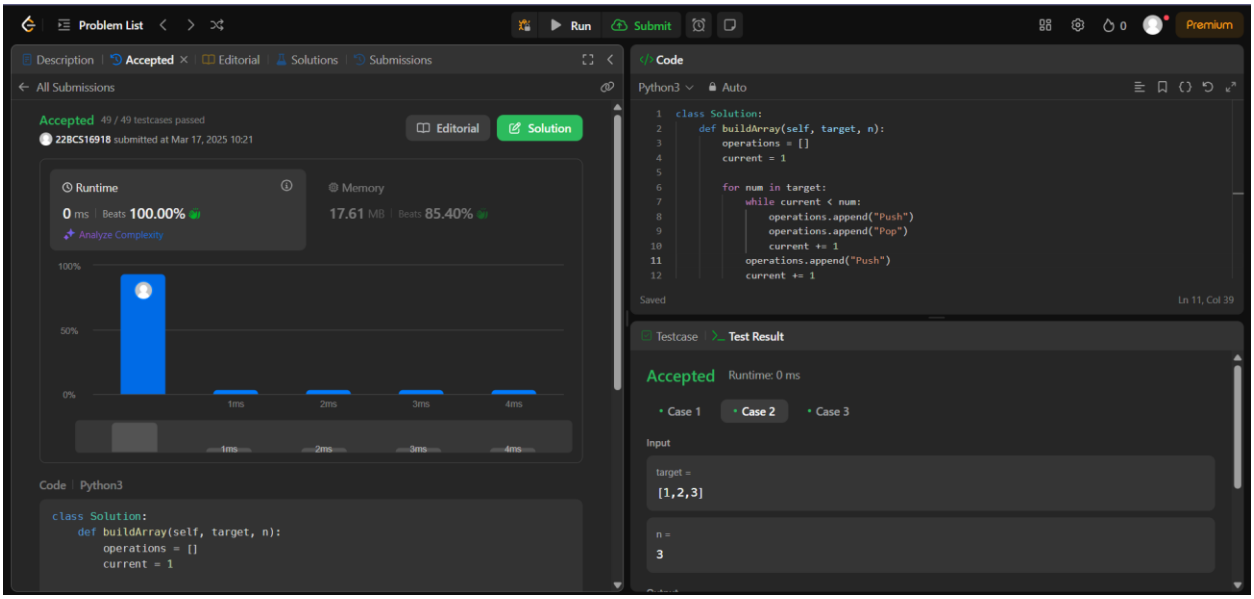
3. Stack using Queue

```
from collections import deque

class MyStack:
    def __init__(self):
        self.q = deque()

    def push(self, x: int) -> None:
        self.q.append(x)
        for _ in range(len(self.q) - 1):
            self.q.append(self.q.popleft())

    def pop(self) -> int:
        return self.q.popleft()

    def top(self) -> int:
        return self.q[0]

    def empty(self) -> bool:
        return not self.q
```



4. Build an array using stack

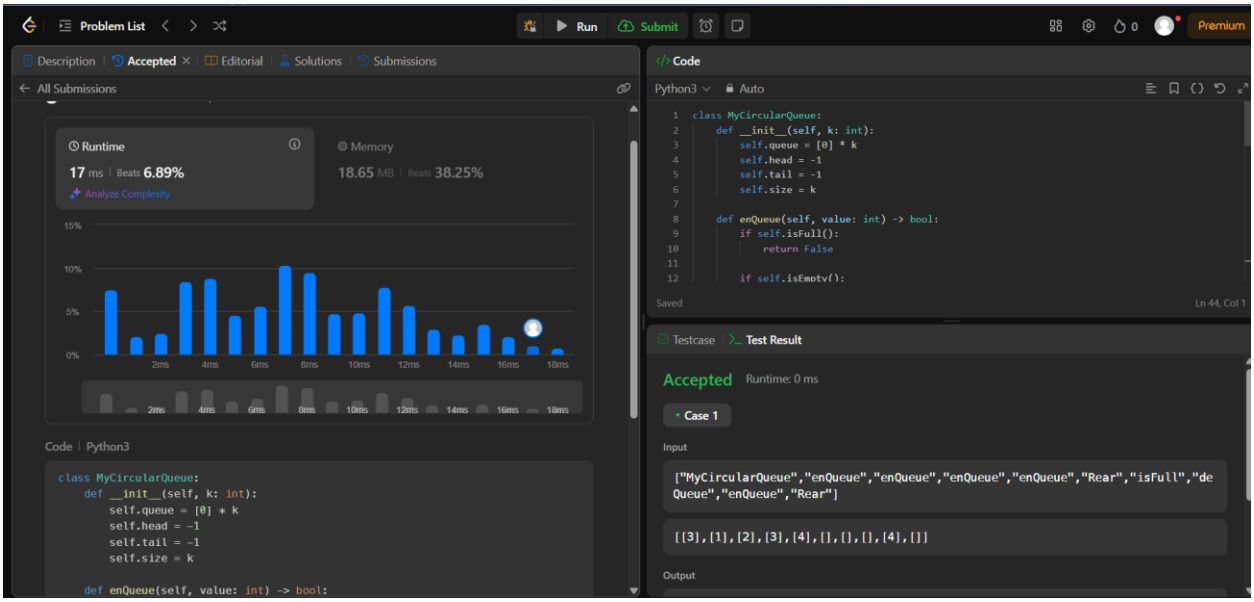
```
class Solution:
    def buildArray(self, target, n):
        operations = []
        current = 1  # Tracks the current number from 1 to n

        for num in target:
            while current < num:  # If there are missing numbers
                operations.append("Push")
                operations.append("Pop")  # Remove unnecessary numbers
                current += 1
            operations.append("Push")  # Push the required number
            current += 1

        return operations
```



5. Circular queue using array

```
class MyCircularQueue:
    def __init__(self, k: int):
        self.queue = [0] * k
        self.head = -1
        self.tail = -1
        self.size = k

    def enQueue(self, value: int) -> bool:
        if self.isFull():
            return False

        if self.isEmpty():
            self.head = 0

        self.tail = (self.tail + 1) % self.size
        self.queue[self.tail] = value
        return True

    def deQueue(self) -> bool:
        if self.isEmpty():
            return False

        if self.head == self.tail:
            self.head = -1
```

```
            self.tail = -1
        else:
            self.head = (self.head + 1) % self.size

        return True

    def Front(self) -> int:
        return -1 if self.isEmpty() else self.queue[self.head]

    def Rear(self) -> int:
        return -1 if self.isEmpty() else self.queue[self.tail]

    def isEmpty(self) -> bool:
        return self.head == -1

    def isFull(self) -> bool:
        return (self.tail + 1) % self.size == self.head
```

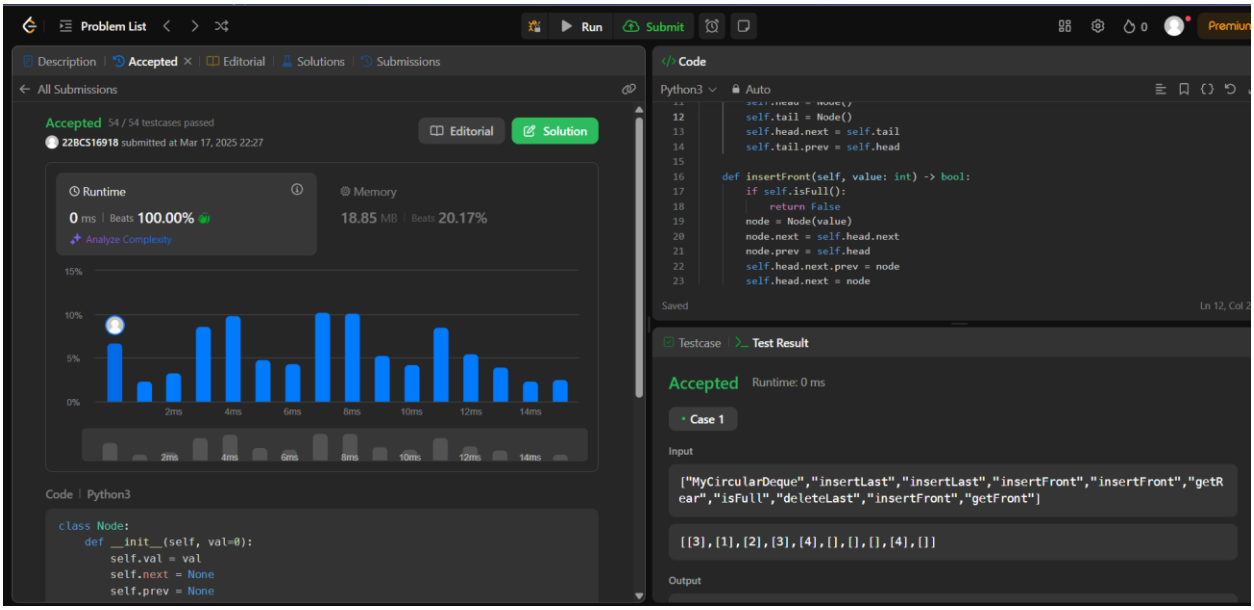


6. Circular deQueue using Linked list

```
class Node:
    def __init__(self, val=0):
        self.val = val
        self.next = None
        self.prev = None

class MyCircularDeque:
```

```
def __init__(self, k: int):
    self.k = k
    self.size = 0
    self.head = Node()
    self.tail = Node()
    self.head.next = self.tail
    self.tail.prev = self.head

def insertFront(self, value: int) -> bool:
    if self.isFull():
        return False
    node = Node(value)
    node.next = self.head.next
    node.prev = self.head
    self.head.next.prev = node
    self.head.next = node
    self.size += 1
    return True

def insertLast(self, value: int) -> bool:
    if self.isFull():
        return False
    node = Node(value)
    node.prev = self.tail.prev
    node.next = self.tail
    self.tail.prev.next = node
    self.tail.prev = node
    self.size += 1
    return True

def deleteFront(self) -> bool:
    if self.isEmpty():
        return False
    self.head.next = self.head.next.next
    self.head.next.prev = self.head
    self.size -= 1
    return True

def deleteLast(self) -> bool:
    if self.isEmpty():
```

```
            return False
        self.tail.prev = self.tail.prev.prev
        self.tail.prev.next = self.tail
        self.size -= 1
        return True

    def getFront(self) -> int:
        return -1 if self.isEmpty() else self.head.next.val

    def getRear(self) -> int:
        return -1 if self.isEmpty() else self.tail.prev.val

    def isEmpty(self) -> bool:
        return self.size == 0

    def isFull(self) -> bool:
        return self.size == self.k
```

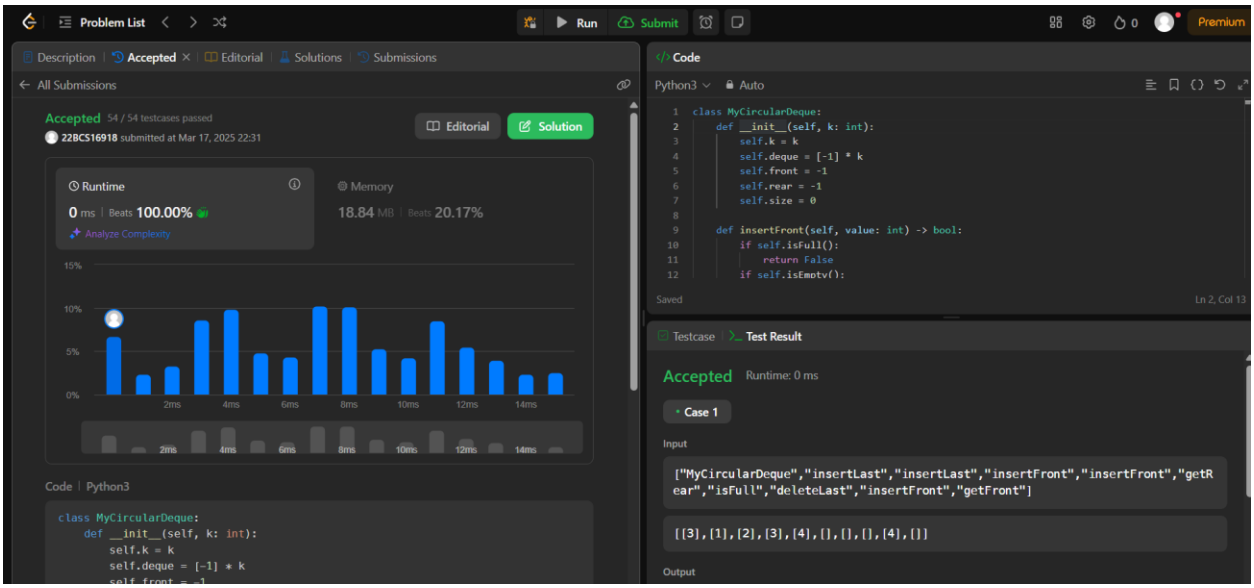


7. Circular Dequeue using Array

```
class MyCircularDeque:
    def __init__(self, k: int):
        self.k = k
        self.deque = [-1] * k  # Fixed-size array
        self.front = -1
        self.rear = -1
        self.size = 0
```

```
def insertFront(self, value: int) -> bool:
    if self.isFull():
        return False
    if self.isEmpty():
        self.front = self.rear = 0
    else:
        self.front = (self.front - 1) % self.k
    self.deque[self.front] = value
    self.size += 1
    return True

def insertLast(self, value: int) -> bool:
    if self.isFull():
        return False
    if self.isEmpty():
        self.front = self.rear = 0
    else:
        self.rear = (self.rear + 1) % self.k
    self.deque[self.rear] = value
    self.size += 1
    return True

def deleteFront(self) -> bool:
    if self.isEmpty():
        return False
    if self.front == self.rear:  # Only one element
        self.front = self.rear = -1
    else:
        self.front = (self.front + 1) % self.k
    self.size -= 1
    return True

def deleteLast(self) -> bool:
    if self.isEmpty():
        return False
    if self.front == self.rear:  # Only one element
        self.front = self.rear = -1
    else:
        self.rear = (self.rear - 1) % self.k
```

```
        self.size -= 1
        return True

    def getFront(self) -> int:
        return -1 if self.isEmpty() else self.deque[self.front]

    def getRear(self) -> int:
        return -1 if self.isEmpty() else self.deque[self.rear]

    def isEmpty(self) -> bool:
        return self.size == 0

    def isFull(self) -> bool:
        return self.size == self.k
```



8. Implement BST (Inorder Traversal) using Stack (Iterative DFS)

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def inorder_traversal(root):
    stack = []
    result = []
    current = root
```

```
    while current or stack:

        while current:
            stack.append(current)
            current = current.left

        current = stack.pop()
        result.append(current.val)

        current = current.right

    return result

# Example
if __name__ == "__main__":
    root = TreeNode(1)
    root.right = TreeNode(2)
    root.right.left = TreeNode(3)

    print(inorder_traversal(root))
```

main.py

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def inorder_traversal(root):
    stack = []
    result = []
    current = root

    while current or stack:

        while current:
            stack.append(current)
            current = current.left

        current = stack.pop()
        result.append(current.val)

        current = current.right

    return result

# Example
if __name__ == "__main__":
    root = TreeNode(1)
```

input

```
[1, 3, 2]


...Program finished with exit code 0
Press ENTER to exit console.
```

9. Implement Graph BFS using Queue

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        node = queue.popleft()
        print(node, end=' ')

        for neighbor in graph.get(node, []):
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)

# Example
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F', 'G'],
    'D': ['B'],
    'E': ['B', 'H'],
    'F': ['C'],
    'G': ['C'],
    'H': ['E']
}
bfs(graph, 'D')
```

```python
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        node = queue.popleft()
        print(node, end=' ')

        for neighbor in graph.get(node, []):
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)

# Example
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F', 'G'],
    'D': ['B'],
    'E': ['B', 'H'],
    'F': ['C'],
    'G': ['C'],
    'H': ['E']
}
```

input

```
D B A E C H F G

...Program finished with exit code 0
Press ENTER to exit console.
```