ASSIGNMENT 2 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 19: Data Structures and Algorithms		
Submission date		**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Truong Tan Phuc	**Student ID**	GCD210070
Class	GCD1101	**Assessor name**	Pham Thanh Son

Student declaration

I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.

Student's signature	Phuc

Grading grid

P4	P5	P6	P7	M4	M5	D3	D4

❐ **Summative Feedback:**	❐ **Resubmission Feedback:**

Grade:	**Assessor Signature:**	**Date:**

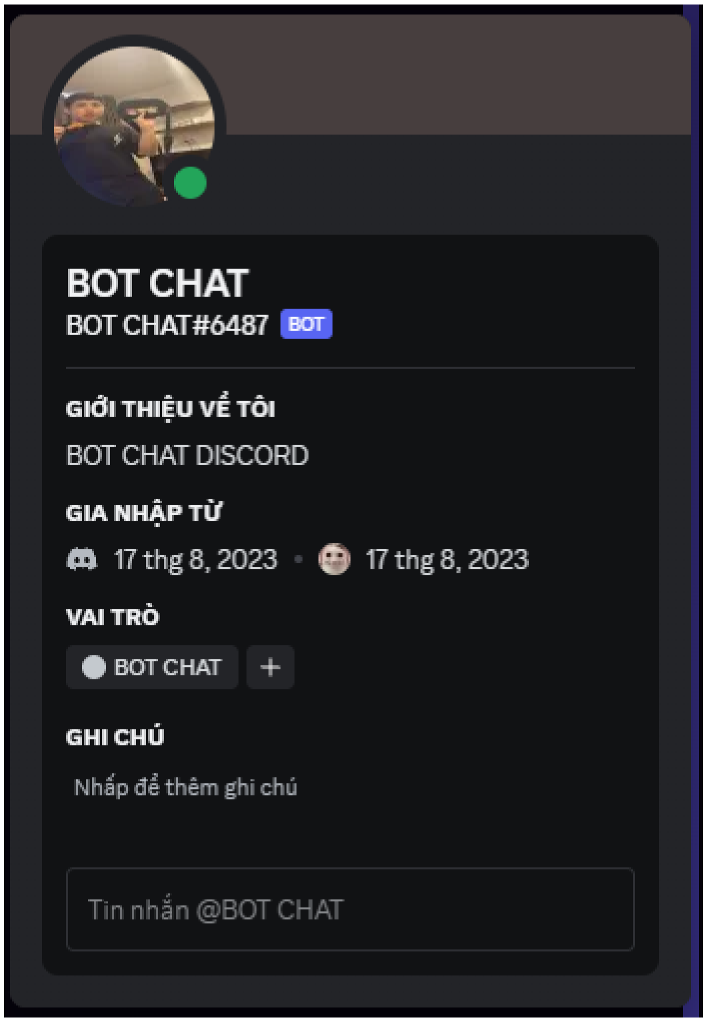
Internal Verifier's Comments:
IV Signature:

Table of Contents

1 IMPLEMENT ADT & ALGORITHMS

1.1 Description of my Bot Chat

1.1.1 Project Description:


BOT CHAT
BOT CHAT#6487 BOT

GIỚI THIỆU VỀ TÔI
BOT CHAT DISCORD

GIA NHẬP TỪ
17 thg 8, 2023 • 17 thg 8, 2023

VAI TRÒ
BOT CHAT +

GHI CHÚ
Nhấp để thêm ghi chú

Tin nhắn @BOT CHAT

Figure 1: CHAT BOT DISCORD

THE DISCORD CHAT BOT with Stack and Queue is a project designed to create an interactive **CHATBOT** within a Discord server. The **CHATBOT** is capable of managing both a message queue and a message stack to facilitate conversations and responses with users. The main purpose of this bot is to provide answers to user questions from a pre-loaded set of responses stored in a queue. Additionally, users can interact with the bot by adding questions to a stack, which will then be paired with responses from the queue for seamless and dynamic conversations.

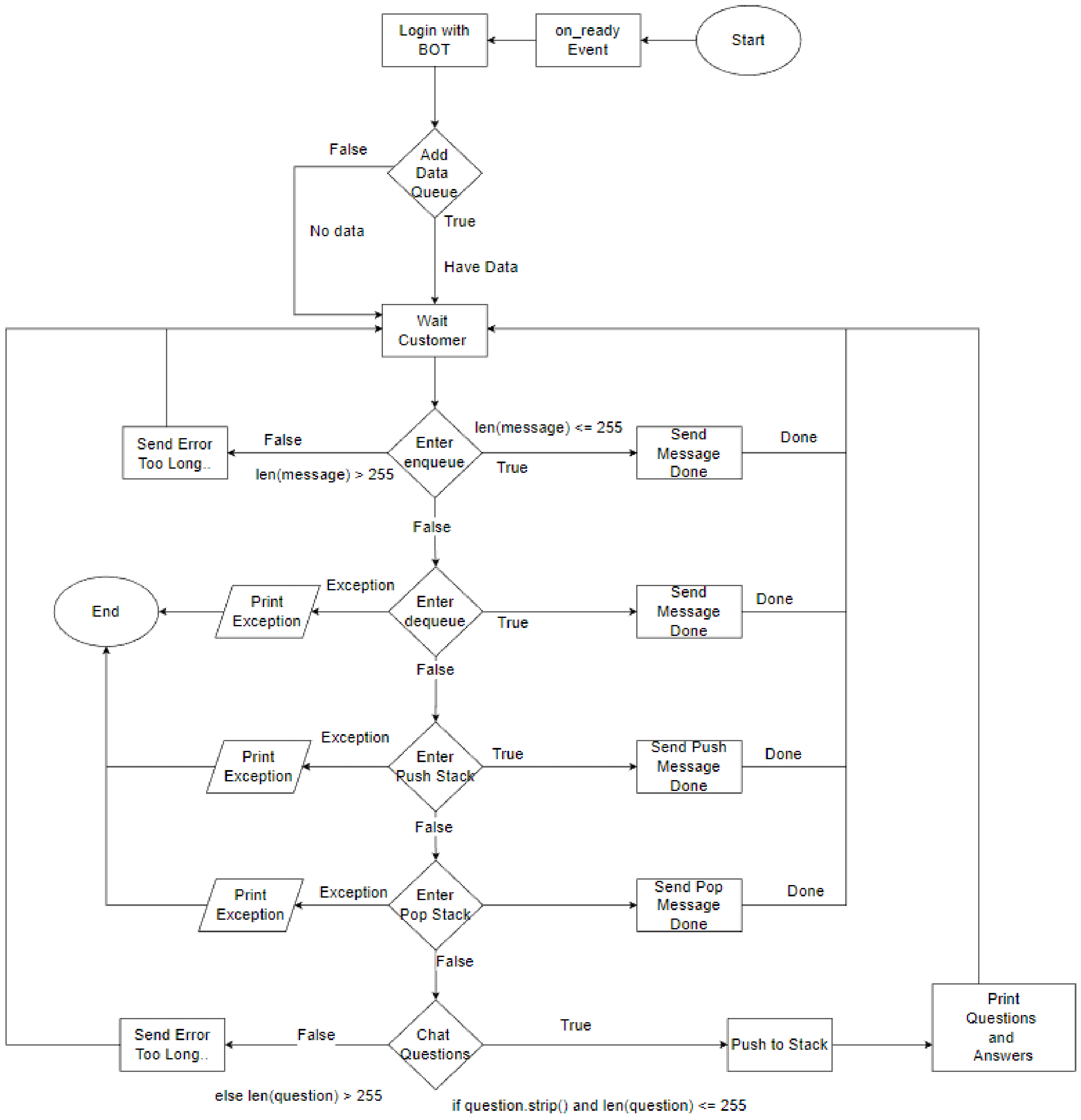


Figure 2: CHAT BOT Flowchart

1.1.2 Reason for Data Structure Selection:

The choice of using both a stack and a queue in this project is based on the specific requirements of managing user interactions and responses:

A, Message Queue:

The message queue is employed to store a collection of pre-defined responses, allowing the bot to access and deliver them to users in a first-in, first-out (FIFO) manner. This is particularly useful for delivering responses to questions sequentially and ensuring that each response is used only once before cycling back to the beginning of the queue. This approach helps maintain a sense of variety in the bot's interactions.

B, Message Stack:

The message stack is used to allow users to input questions in a last-in, first-out (LIFO) manner. This means that the most recently asked question will be paired with the next response from the queue. Using a stack for user input enables the bot to provide responses in the order that the questions were asked, creating a more dynamic and interactive conversation experience.

By combining the stack and queue data structures, the project aims to create a bot that can engage in conversations with users while providing structured and organized responses from the predefined set of messages. The use of these data structures enhances the bot's ability to manage conversations, handle user input, and provide engaging interactions within the Discord server.

1.2 ADT (P4, P5)

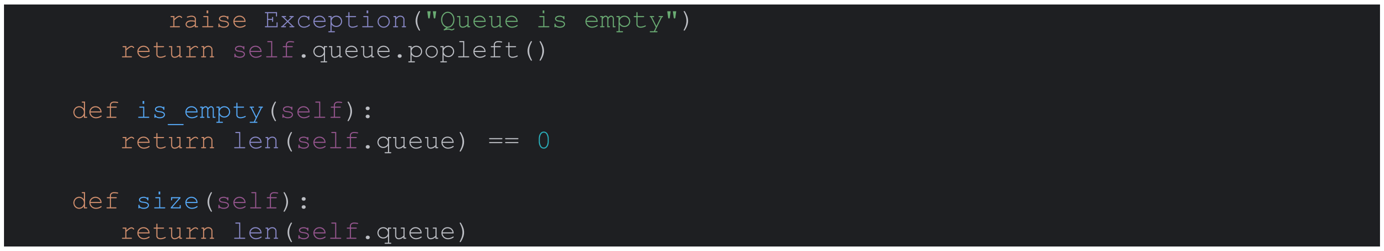
1.2.1 Explain your implementation of the data structure (P4)

A, MessageQueue Implementation:

```
class MessageQueue:
    def __init__(self):
        self.queue = deque()

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

    def dequeue(self):
        if self.is_empty():
```



```
            raise Exception("Queue is empty")
        return self.queue.popleft()

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

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

Figure 3: MessageQueue Code

Explanation:

The **MessageQueue** class is implemented using the deque (double-ended queue) data structure from the collections module. This data structure is chosen because it supports efficient enqueueing and dequeuing operations from both ends of the queue, making it suitable for managing automated responses in a conversation.

- __init__(self): The constructor initializes an empty **deque** when an instance of **MessageQueue** is created.
- **enqueue(self, message)**: This method adds a new message to the end of the queue using the **append()** method of the **deque**. This reflects the behavior of adding new messages to the end of the conversation queue.
- **dequeue(self):** This method removes and returns the message at the front of the queue using the **popleft()** method of the deque. It raises an exception if the queue is empty.
- **is_empty(self):** This method checks if the queue is empty by comparing its length to zero.
- **size(self):** This method returns the current size of the queue, indicating the number of messages in the queue.

Here's a step-by-step explanation of how the MessageQueue works:

Initialization (__init__): When an instance of MessageQueue is created, the constructor initializes an empty deque called self.queue.

Enqueueing Messages (enqueue): The enqueue method is used to add a new message to the end of the queue. This is done using the append method of the deque, which reflects the behavior of adding new messages to the end of a conversation queue.

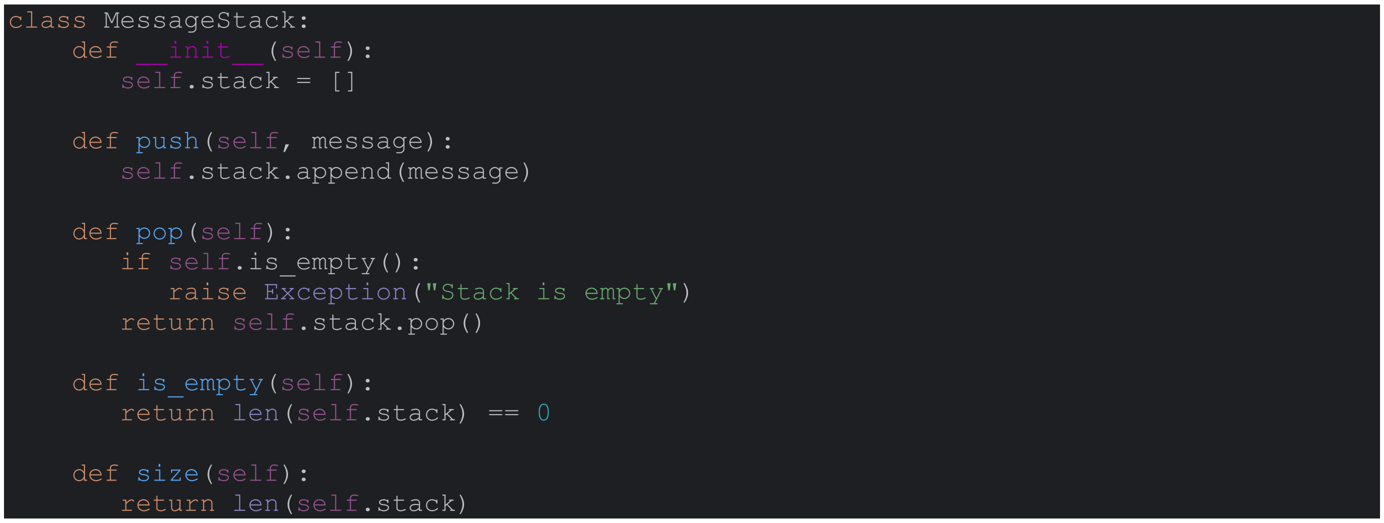
Dequeuing Messages (dequeue): The dequeue method is used to remove and return the message from the front of the queue. It uses the popleft method of the deque to ensure that the messages are processed in the order they were added (FIFO). If an attempt is made to dequeue from an empty queue, an exception is raised with the error message "Queue is empty."

Checking if Queue is Empty (is_empty): The is_empty method checks whether the queue is empty by comparing the length of the queue to zero. If the queue has no messages, this method will return True.

Getting Queue Size (size): The size method returns the current size of the queue, indicating the number of messages it currently holds.

Overall, the MessageQueue class in this code manages the orderly processing of automated responses. Messages are enqueued (added) to the end of the queue and dequeued (removed) from the front of the queue, ensuring that messages are processed in the order they were received. If an attempt is made to dequeue from an empty queue, an exception is raised to handle this error scenario.

B, MessageStack Implementation:



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

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

    def pop(self):
        if self.is_empty():
            raise Exception("Stack is empty")
        return self.stack.pop()

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

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

Figure 4: Message Stack

Explanation:

The **MessageStack** class is implemented using a Python list to mimic the behavior of a stack. The stack data structure is used to manage messages pushed by administrators, allowing for last-in, first-out (LIFO) processing.

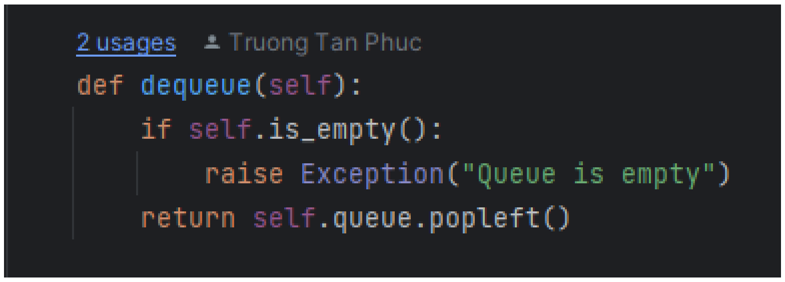
- **__init__(self):** The constructor initializes an empty list when an instance of **MessageStack** is created.
- **push(self, message):** This method adds a new message to the top of the stack using the **append()** method of the list. This simulates the behavior of adding new messages to the top of the stack.
- **pop(self):** This method removes and returns the message from the top of the stack using the **pop()** method of the list. It raises an exception if the stack is empty.
- **is_empty(self):** This method checks if the stack is empty by comparing its length to zero.

- **size(self):** This method returns the current size of the stack, indicating the number of messages in the stack.

In summary, **the MessageQueue** and **MessageStack** implementations provide the necessary functionality to manage messages in a chat bot context. The **MessageQueue** ensures orderly processing of automated responses using a deque, while **the MessageStack** enables administrators to manage and present messages using a list-based stack. These implementations align with the characteristics of queues and stacks, respectively, and contribute to the overall functionality of the chat bot application.

1.2.2 Explain how you handle error by exception (P5)

A, Handling Errors by Exception in MessageQueue:



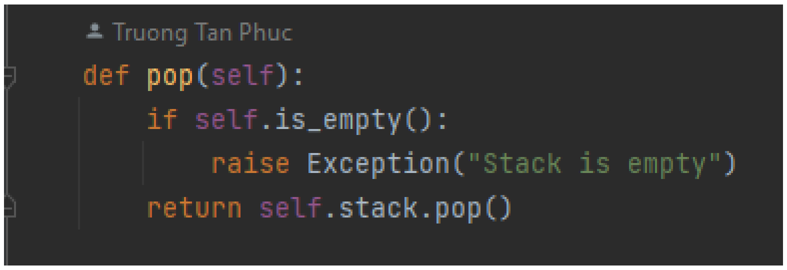
```
2 usages  Truong Tan Phuc
def dequeue(self):
    if self.is_empty():
        raise Exception("Queue is empty")
    return self.queue.popleft()
```

Figure 5: MessageQueue Handling Errors

Explanation:

In the MessageQueue class, error handling is achieved using custom exceptions. When the **dequeue()** method is called on an empty queue, it raises an exception with a specific error message indicating that the queue is empty. This allows for better control and more informative error messages when something goes wrong in the code.

B, Handling Errors by Exception in MessageStack:



```
Truong Tan Phuc
def pop(self):
    if self.is_empty():
        raise Exception("Stack is empty")
    return self.stack.pop()
```

Figure 6: MessageStack Handling Errors

Explanation:

Similarly, in the MessageStack class, the pop method is responsible for removing and returning a message from the top of the stack. If the stack is empty and a pop is attempted, it would result in an error. To manage this situation, an exception is raised with the message "Stack is empty". This custom exception serves the purpose of clearly indicating the issue and enabling developers or users to understand why the error occurred. The raised exception is handled in the code by catching it and responding with relevant information.

Advantages of Error Handling with Exceptions:

- **Clear Identification:** Custom exceptions provide specific error messages, making it easier to identify the nature of the error and where it occurred in the code.
- **Separation of Concerns:** Using exceptions separates error handling logic from the main logic of the methods. This results in cleaner and more readable code.
- **Structured Flow:** By raising exceptions in specific cases, you can control the flow of the program and ensure that errors are handled gracefully.

Overall Impact:

In the provided code, handling errors by raising custom exceptions enhances the code's robustness and user-friendliness. When an operation is attempted on an empty queue or stack, the corresponding exceptions are raised, giving users clear information about what went wrong. This makes debugging and troubleshooting more efficient and contributes to a smoother user experience.

1.2.3 Sourcecode:

```
import discord
from discord.ext import commands
```

```python
from collections import deque
import random
from discord import Intents

intents = discord.Intents.default()
intents.typing = True
intents.presences = True
intents.message_content = True

bot = commands.Bot(command_prefix="/", intents=intents)


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

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

    def dequeue(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        return self.queue.popleft()

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

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


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

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

    def pop(self):
        if self.is_empty():
            raise Exception("Stack is empty")
        return self.stack.pop()

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

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


queue = MessageQueue()
stack = MessageStack()

# Đẩy nội dung từ messager_auto.txt vào queue
with open("messager_auto.txt", "r", encoding="utf-8") as auto_file:
    for line in auto_file:
        line = line.strip()
        queue.enqueue(line)


@bot.command(name='enqueue')
async def enqueue_message(ctx, *, message):
    if len(message) > 255:
        await ctx.send("Câu hỏi quá dài. Vui lòng nhập câu hỏi ngắn hơn.")
        return
    queue.enqueue(message)
    await ctx.send(f'Message "{message}" has been enqueued.')


@bot.command(name='dequeue')
async def dequeue_message(ctx):
    try:
        message = queue.dequeue()
        await ctx.send(f'Dequeued message: {message}')
    except Exception as e:
        await ctx.send(str(e))


@bot.command(name='push')
async def push_message(ctx, *, message):
    if len(message) > 255:
        await ctx.send("Câu hỏi quá dài. Vui lòng nhập câu hỏi ngắn hơn.")
        return
    else:
        stack.push(message)
        await ctx.send(f'Message "{message}" has been pushed to the stack.')


@bot.command(name='pop')
async def pop_message(ctx):
    try:
        message = stack.pop()
        await ctx.send(f'Popped message: {message}')
    except Exception as e:
        await ctx.send(str(e))


@bot.event
async def on_ready():
    print(f'Logged in as {bot.user.name}')


@bot.command(name='chat')
async def chat(ctx):
    user_message = ctx.message.content.replace("/chat", "").strip()
    user_questions = user_message.split("?")
    for question in user_questions:
        if question.strip() and len(question) <= 255:
            stack.push(question.strip())
        elif len(question) > 255:
            await ctx.send("Câu hỏi quá dài. Vui lòng nhập câu hỏi ngắn hơn.")
            break

    while not stack.is_empty():
      if not queue.is_empty():
         stacked_message = stack.pop()
         queue_message = queue.dequeue()
         await ctx.send(f' **Question:** {stacked_message} \n**Answer:**
{queue_message}\n-------------------')
      else:
         await ctx.send(f'Vui lòng đợi admin thêm câu trả lời.')


bot.run('BOT_TOKEN')
```

Figure 7: Sourcecode for CHAT BOT DISCORD SERVER

A, Explain:

- **Code Import Library:**

```
import discord
from discord.ext import commands
from collections import deque
import random
from discord import Intents

intents = discord.Intents.default()
intents.typing = True
intents.presences = True
intents.message_content = True

bot = commands.Bot(command_prefix="/", intents=intents)
```

Figure 8: Start Code

In this section, necessary libraries are imported. The collections module is used for the deque data structure. The discord and discord.ext.commands modules are imported to interact with the Discord API. The Intents class is used to define the level of detail about Discord events that the bot should be able to access. An instance of the Bot class is created, which will represent the Discord bot with a specified command prefix and intents.

- **Message Queue Class:**

```
class MessageQueue:
   def __init__(self):
      self.queue = deque()

   def enqueue(self, message):
      self.queue.append(message)
```

```
    def dequeue(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        return self.queue.popleft()

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

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

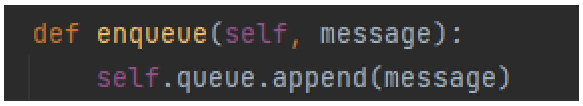
Figure 9: MessageQueue Class

This is the implementation of the MessageQueue class using a deque, which was explained in previous responses. It handles enqueuing and dequeuing messages in a first-in, first-out (FIFO) manner.

```
def __init__(self):
    self.queue = deque()
```

Figure 10: __init__ function

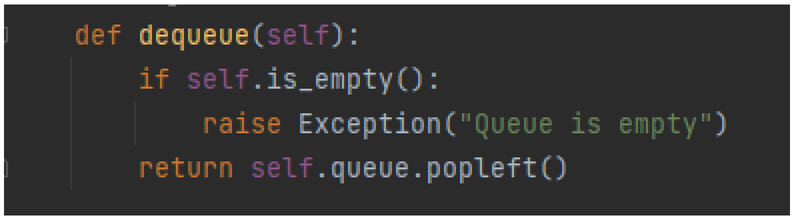
The constructor initializes an instance of the MessageQueue class. It creates an empty deque object called queue, which will be used to store the messages in the queue.



```
def enqueue(self, message):
    self.queue.append(message)
```

Figure 11: enqueuer function

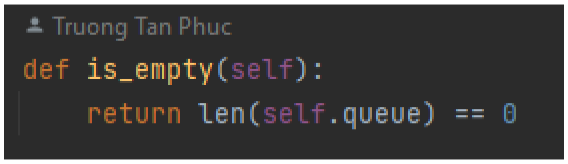
This method adds a message to the end of the queue. It uses the append method of the deque to insert the provided message at the end of the queue. The new message is enqueued in a first-in, first-out (FIFO) manner.



```
def dequeue(self):
    if self.is_empty():
        raise Exception("Queue is empty")
    return self.queue.popleft()
```

Figure 12: dequeue function

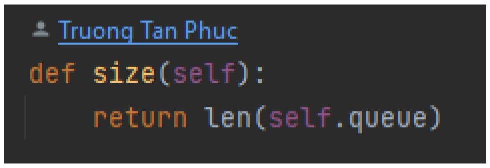
This method removes and returns the message from the front of the queue. It uses the popleft method of the deque to pop the first message in the queue, ensuring that messages are processed in the order they were added (FIFO). If the queue is empty, it raises an exception with the message "Queue is empty."



```
Truong Tan Phuc
def is_empty(self):
    return len(self.queue) == 0
```

Figure 13: is_empty function

This method checks if the queue is empty. It returns True if the length of the queue is 0, indicating that there are no messages in the queue. Otherwise, it returns False.



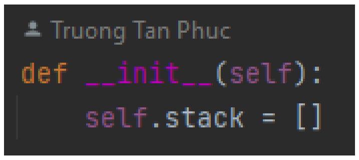
```
Truong Tan Phuc
def size(self):
    return len(self.queue)
```

Figure 14: size function

This method returns the current size of the queue, indicating the number of messages it holds. It does this by returning the length of the deque, which corresponds to the number of messages currently enqueued.

The MessageQueue class implements a basic queue data structure using a deque from the collections module. It provides methods for enqueueing, dequeueing, checking if the queue is empty, and getting the size of the queue. This allows messages to be managed in an ordered manner, ensuring that the first message enqueued is the first to be dequeued.

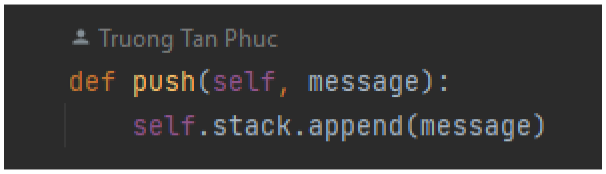
- **MessageStack:**



```
Truong Tan Phuc
def __init__(self):
    self.stack = []
```

Figure 15: __init__ function

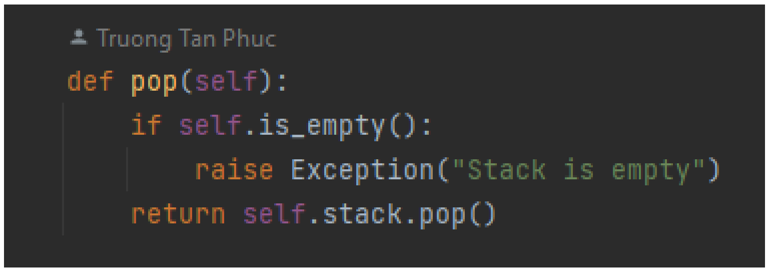
The constructor initializes an instance of the **MessageStack** class. It creates an empty list called **stack**, which will be used to store the messages in the stack.



```
Truong Tan Phuc
def push(self, message):
    self.stack.append(message)
```

Figure 16: push function

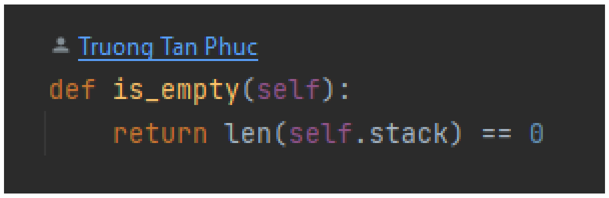
This method adds a message to the top of the stack. It uses the append method of the list to insert the provided message at the end of the stack, simulating the behavior of adding messages to the top of a stack data structure.



```
Truong Tan Phuc
def pop(self):
    if self.is_empty():
        raise Exception("Stack is empty")
    return self.stack.pop()
```

Figure 17: pop function

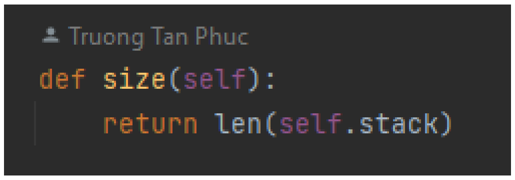
This method removes and returns the message from the top of the stack. It uses the **pop** method of the list to remove and return the last message in the stack, simulating the behavior of removing the top element from a stack. If the stack is empty and a pop is attempted, it raises an exception with the message "Stack is empty."



```
Truong Tan Phuc
def is_empty(self):
    return len(self.stack) == 0
```

Figure 18: is_empty

This method checks if the stack is empty. It returns **True** if the length of the stack is 0, indicating that there are no messages in the stack. Otherwise, it returns **False**.



```
Truong Tan Phuc
def size(self):
    return len(self.stack)
```

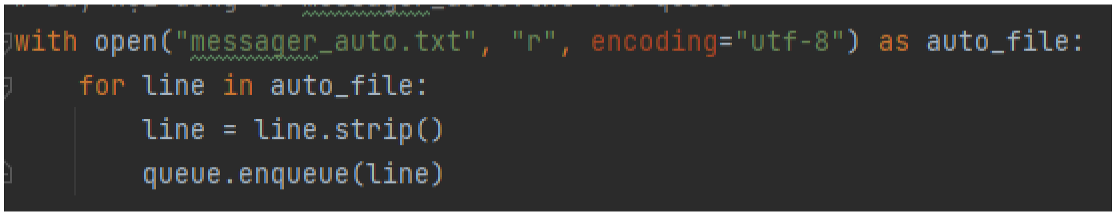
Figure 19: size

This method returns the current size of the stack, indicating the number of messages it holds. It does this by returning the length of the list, which corresponds to the number of messages currently in the stack.

Summary:

The **MessageStack** class implements a basic stack data structure using a Python list. It provides methods for pushing, popping, checking if the stack is empty, and getting the size of the stack. This allows messages to be managed in a last-in, first-out (LIFO) manner, similar to how elements are managed in a real-world stack.

- **Read file messanger:**



```
with open("messager_auto.txt", "r", encoding="utf-8") as auto_file:
    for line in auto_file:
        line = line.strip()
        queue.enqueue(line)
```

Figure 20: Read file code

with open("messager_auto.txt", "r", encoding="utf-8") as auto_file:

- This line opens the "messager_auto.txt" file in read mode ("r") with the specified UTF-8 encoding. The with statement ensures that the file is properly closed after it's been read.

for line in auto_file:

- This loop iterates through each line in the opened file (auto_file).

line = line.strip():

- The **strip()** method is used to remove leading and trailing whitespace characters (such as spaces and newlines) from each line. This ensures that any extra whitespace is removed from the lines before they are processed.

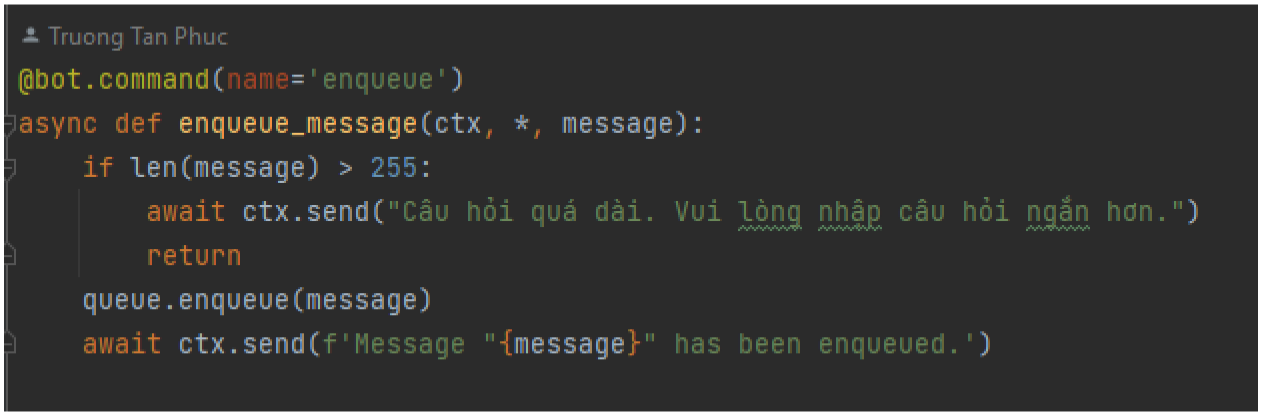
queue.enqueue(line):

- This line enqueues the cleaned-up line into the queue using the enqueue method of the MessageQueue class. This means that each line from the file, after being stripped of whitespace, is added to the end of the message queue.

Summary:

The provided code block reads each line from the "messager_auto.txt" file, removes any leading or trailing whitespace from the line, and then enqueues the cleaned-up line into the message queue using the enqueue method. This process fills the message queue with pre-defined messages from the text file, ensuring that these messages are available for automated responses in the chatbot's conversation flow.

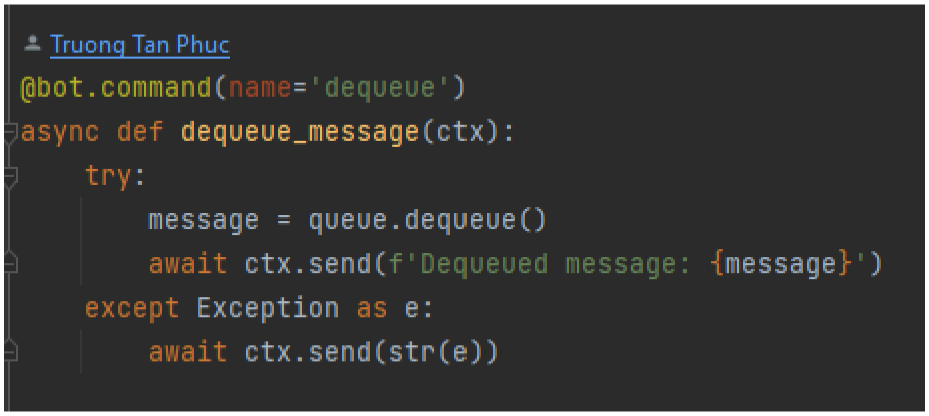
- **BOT CHAT Command:**



```
Truong Tan Phuc
@bot.command(name='enqueue')
async def enqueue_message(ctx, *, message):
    if len(message) > 255:
        await ctx.send("Câu hỏi quá dài. Vui lòng nhập câu hỏi ngắn hơn.")
        return
    queue.enqueue(message)
    await ctx.send(f'Message "{message}" has been enqueued.')
```

Figure 21: enqueue command function

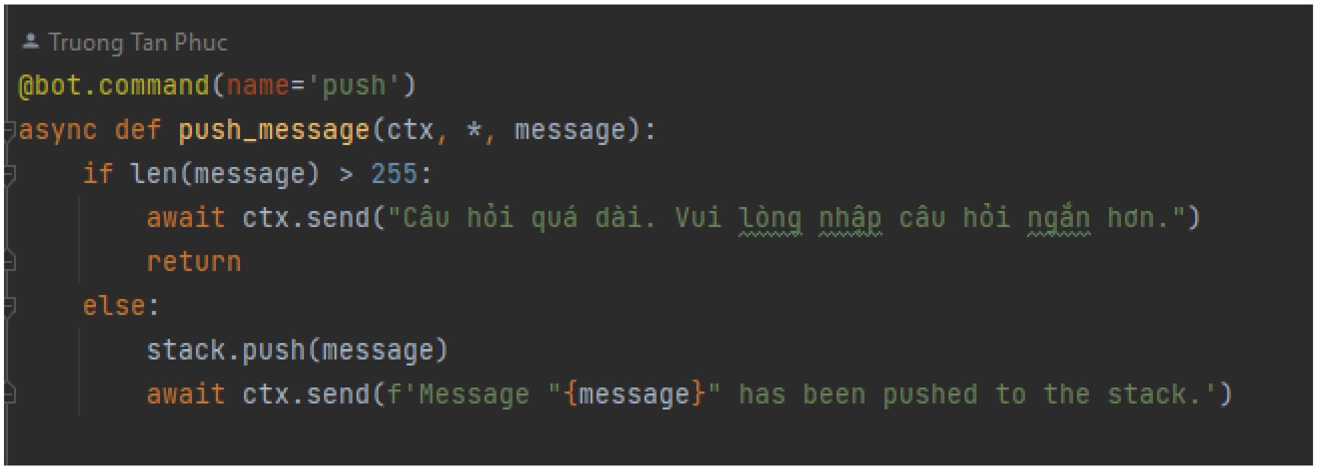
- This command function is named enqueue_message.
- It takes the user's message as an argument.
- If the length of the message is more than 255 characters, it sends a response saying that the question is too long.
- If the message length is within the limit, it enqueues the message using the enqueue method of the MessageQueue class.
- It sends a response confirming that the message has been enqueued.



```
Truong Tan Phuc
@bot.command(name='dequeue')
async def dequeue_message(ctx):
    try:
        message = queue.dequeue()
        await ctx.send(f'Dequeued message: {message}')
    except Exception as e:
        await ctx.send(str(e))
```

Figure 22: dequeue function command

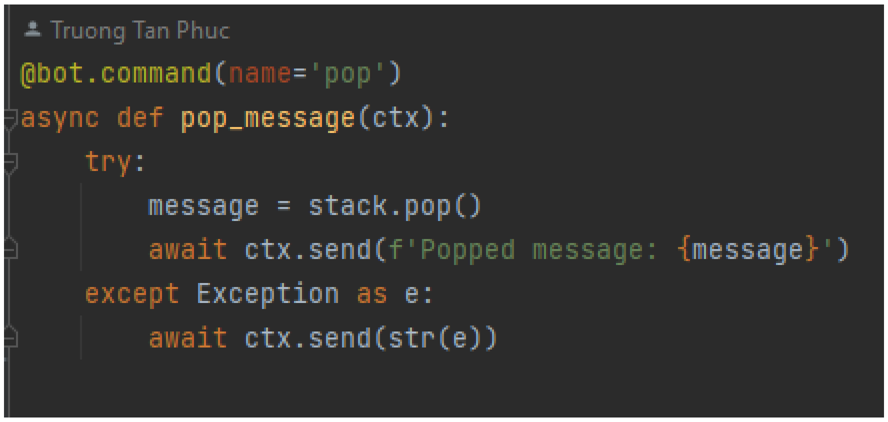
- This command function is named dequeue_message.
- It tries to dequeue a message using the dequeue method of the MessageQueue class.
- If the queue is empty, an exception is caught and a response is sent indicating that the queue is empty.
- If a message is successfully dequeued, the message is sent to the channel.



```
Truong Tan Phuc
@bot.command(name='push')
async def push_message(ctx, *, message):
    if len(message) > 255:
        await ctx.send("Câu hỏi quá dài. Vui lòng nhập câu hỏi ngắn hơn.")
        return
    else:
        stack.push(message)
        await ctx.send(f'Message "{message}" has been pushed to the stack.')
```

Figure 23: push function command

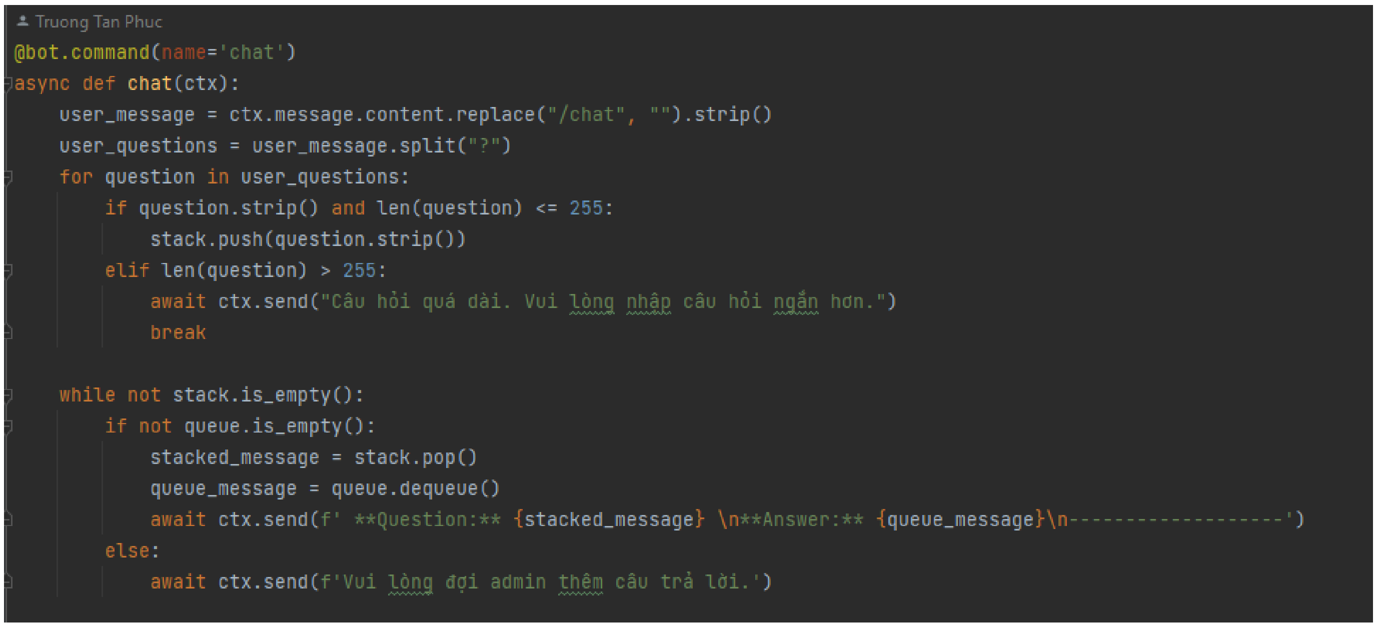
- This command function is named push_message.
- It takes the user's message as an argument.
- If the length of the message is more than 255 characters, it sends a response indicating that the question is too long.
- If the message length is within the limit, it pushes the message onto the stack using the push method of the MessageStack class.
- It sends a response confirming that the message has been pushed to the stack.



```
Truong Tan Phuc
@bot.command(name='pop')
async def pop_message(ctx):
    try:
        message = stack.pop()
        await ctx.send(f'Popped message: {message}')
    except Exception as e:
        await ctx.send(str(e))
```

Figure 24: pop function command

- This command function is named pop_message.
- It tries to pop a message from the stack using the pop method of the MessageStack class.
- If the stack is empty, an exception is caught and a response is sent indicating that the stack is empty.
- If a message is successfully popped, the message is sent to the channel.



```
Truong Tan Phuc
@bot.command(name='chat')
async def chat(ctx):
    user_message = ctx.message.content.replace("/chat", "").strip()
    user_questions = user_message.split("?")
    for question in user_questions:
        if question.strip() and len(question) <= 255:
            stack.push(question.strip())
        elif len(question) > 255:
            await ctx.send("Câu hỏi quá dài. Vui lòng nhập câu hỏi ngắn hơn.")
            break

    while not stack.is_empty():
        if not queue.is_empty():
            stacked_message = stack.pop()
            queue_message = queue.dequeue()
            await ctx.send(f' **Question:** {stacked_message} \n**Answer:** {queue_message}\n-------------------')
        else:
            await ctx.send(f'Vui lòng đợi admin thêm câu trả lời.')
```

Figure 25: chat functiom command

- This command function is named chat.
- It processes a chat message from the user.
- It splits the user's message into separate questions using the split("?") method.
- For each question, it checks if it's not empty and its length is within the limit. If so, it pushes the question onto the stack.
- If a question is too long, it sends a response indicating that the question is too long and exits the loop.
- After processing the questions, it enters a loop and while the stack is not empty and the queue is not empty, it pairs and sends a question from the stack with an answer from the queue.
- If the stack is not empty but the queue is empty, it sends a response indicating that the user needs to wait for an admin to add more answers.

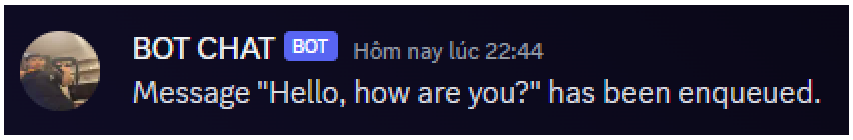
Summary:

The provided code defines command functions for interacting with the message queue and stack. It allows users to enqueue, dequeue, push, and pop messages, as well as simulate a conversation using a combination of messages from the stack and queue.

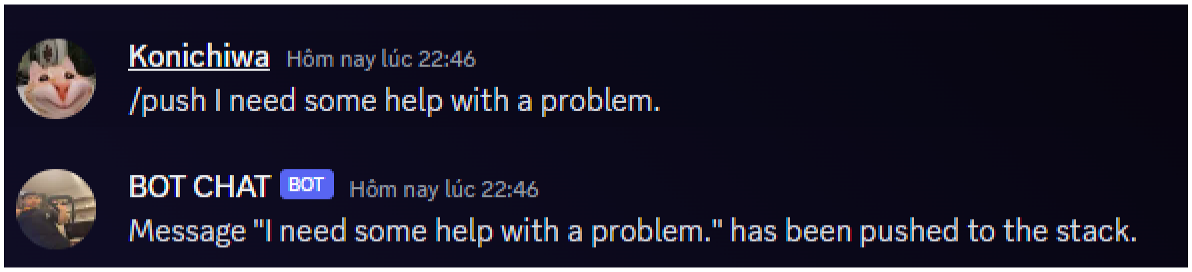
1.2.4 TEST:

TEST CASE	INPUT DATA	EXPECTED OUTPUT	ACTUAL OUTPUT	RESULT
1	/enqueue Hello, how are you?	Message "Hello, how are you?" has been enqueued.		PASS
2	/dequeue	Exception message: Queue is empty.		PASS
3	/push I need some help with a problem.	Message "I need some help with a problem." has been pushed to the stack.		PASS
4	/push This is another very long message that exceeds the character limit and should not be added to the stack.	Câu hỏi quá dài. Vui lòng nhập câu hỏi ngắn hơn.		PASS
5	/pop	Exception message: Stack is empty.		PASS
6	/chat How are you? What's your name?	A combination of questions and answers from the stack and queue.		PASS
7	/chat This is a very long question that exceeds the character limit and should not be added to the stack.	Response: Câu hỏi quá dài. Vui lòng nhập câu hỏi ngắn hơn.		PASS
8	/chat How are you?	Response: Vui lòng đợi admin thêm câu trả lời.		PASS

Result 1:

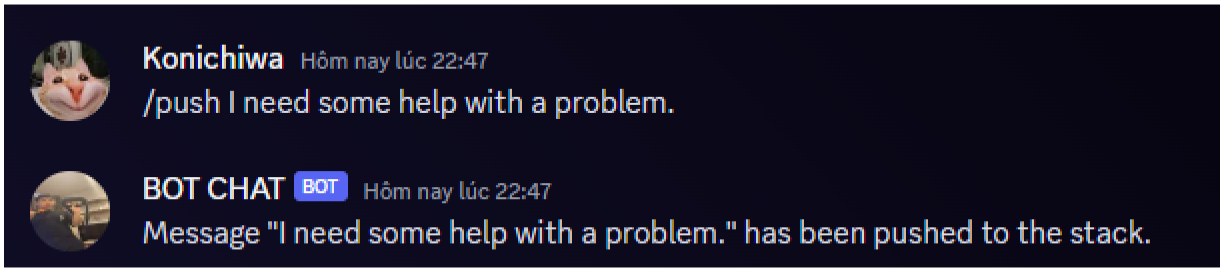

BOT CHAT BOT Hôm nay lúc 22:44
Message "Hello, how are you?" has been enqueued.

Result 2:


Konichiwa Hôm nay lúc 22:46
/push I need some help with a problem.

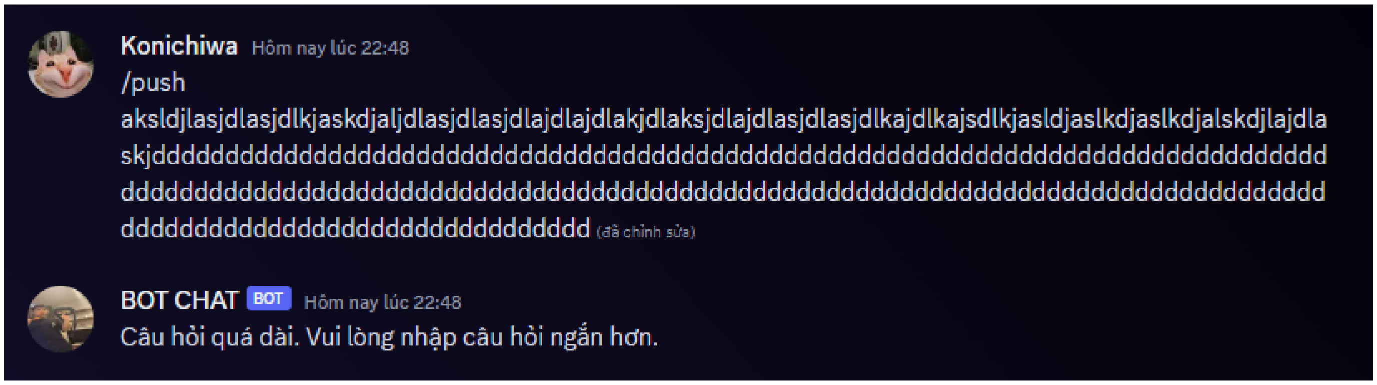
BOT CHAT BOT Hôm nay lúc 22:46
Message "I need some help with a problem." has been pushed to the stack.

Result 3:


Konichiwa Hôm nay lúc 22:47
/push I need some help with a problem.

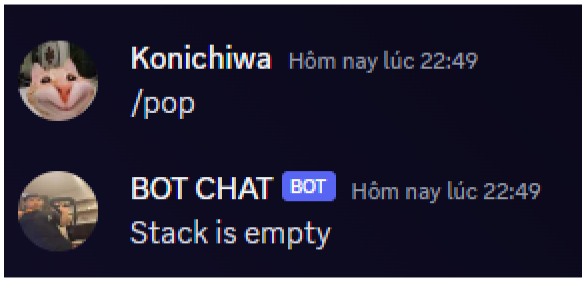
BOT CHAT BOT Hôm nay lúc 22:47
Message "I need some help with a problem." has been pushed to the stack.

Result 4:


Konichiwa Hôm nay lúc 22:48
/push
aksldjlasjdlasjdlkjaskdjaljdlasjdlasjdlajdlajdlakjdlaksjdlajdlasjdlasjdlkajdlkajsdlkjasldjaslkdjaslkdjalskdjlajdla
skjdddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddd (đã chỉnh sửa)

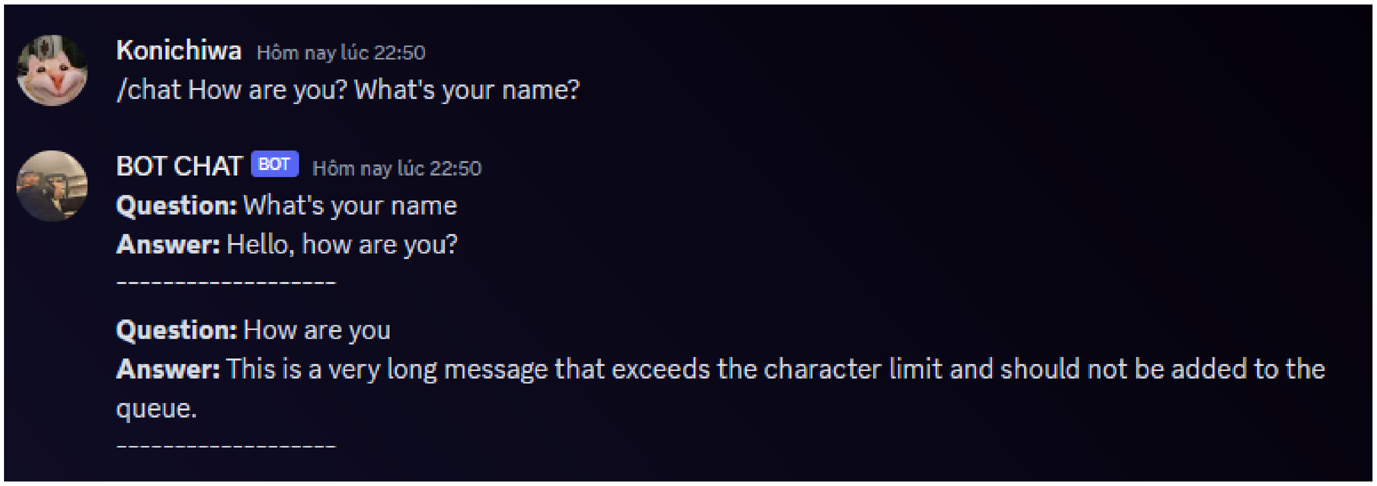
BOT CHAT BOT Hôm nay lúc 22:48
Câu hỏi quá dài. Vui lòng nhập câu hỏi ngắn hơn.

Result 5:


Konichiwa Hôm nay lúc 22:49
/pop

BOT CHAT BOT Hôm nay lúc 22:49
Stack is empty

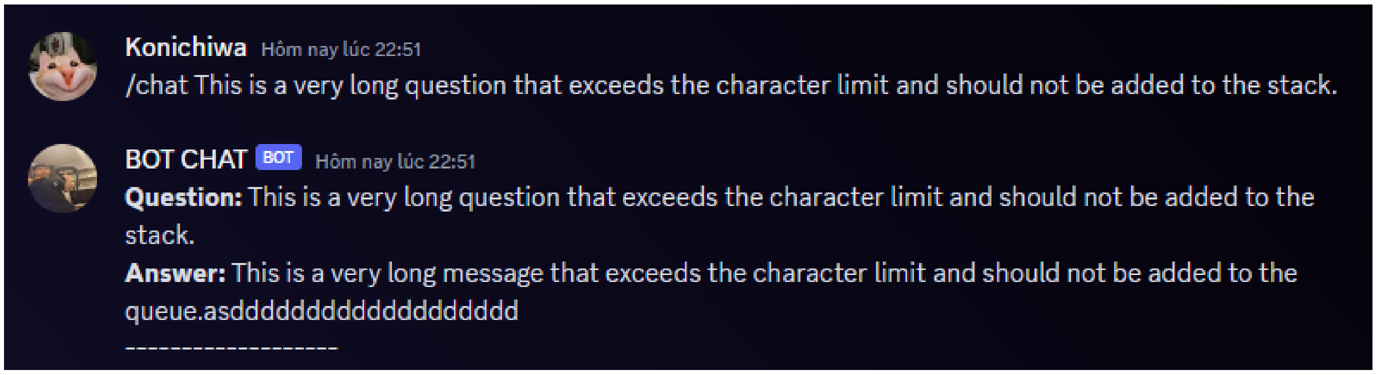
Result 6:


Konichiwa Hôm nay lúc 22:50
/chat How are you? What's your name?

BOT CHAT BOT Hôm nay lúc 22:50
Question: What's your name
Answer: Hello, how are you?

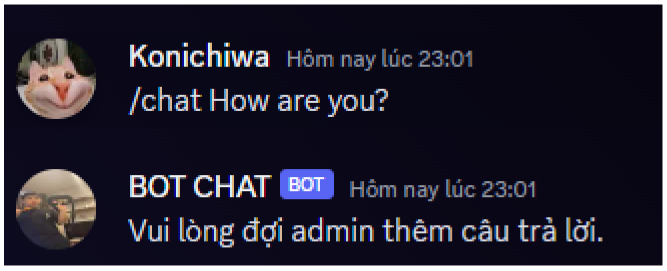
Question: How are you
Answer: This is a very long message that exceeds the character limit and should not be added to the queue.

Result 7:


Konichiwa Hôm nay lúc 22:51
/chat This is a very long question that exceeds the character limit and should not be added to the stack.

BOT CHAT BOT Hôm nay lúc 22:51
Question: This is a very long question that exceeds the character limit and should not be added to the stack.
Answer: This is a very long message that exceeds the character limit and should not be added to the queue.asdddddddddddddddddd
----

Result 8:


Konichiwa Hôm nay lúc 23:01
/chat How are you?

BOT CHAT BOT Hôm nay lúc 23:01
Vui lòng đợi admin thêm câu trả lời.

1.3 Big O (P6)

1.3.1 Asymptotic analysis

We are all familiar with the concept that arranging data using data structures is an efficient approach, and this efficiency can be evaluated in terms of time and space usage. To evaluate the effectiveness of data organization, I employ asymptotic analysis. This technique involves establishing a mathematical limit for the runtime performance of an algorithm. It provides a solution to the challenges encountered in algorithm analysis as discussed earlier.

Within asymptotic analysis, we assess how the time (or space) utilized by a method increases in relation to the size of the input. Ultimately, the optimal data structure is the one that minimizes the time and memory space required to perform all its operations.

The processing time required for an algorithm is typically divided into three categories:

- **Best case:** The algorithm must be processed in the shortest amount of time possible.
- **Average case:** The average time required to run the method.
- **Worst-case:** The maximum amount of time required to process the algorithm.

1.3.2 Asymptotic notations

An algorithm's asymptotic notation is a mathematical representation used to depict an algorithm's complexity by defining the running time of an algorithm when the input trends towards a certain value body or a limit value.

There are three forms of data structure asymptotic notation.

- Big-O Notation
- Big-Ω Notation
- Big Theta

1.3.3 Big O Notation

One approach to assess the efficiency of an algorithm involves the utilization of Big O Notation. As the input size grows, this notation calculates the duration it takes for your function to execute or how the function's scalability performs. Efficiency is evaluated in terms of both temporal complexity and spatial complexity. The temporal complexity quantifies the execution duration of the function in computational steps, while the spatial complexity pertains to the amount of memory the function consumes. To exemplify the concept of time complexity, this blog will employ two search algorithms. The algorithm's upper limit, often denoted as Big O, is occasionally utilized to characterize its performance in handling worst-case scenarios. In contrast, the best-case scenario, where the item is found in the initial pass, doesn't offer substantial insights. Prioritizing the worst-case scenario eliminates uncertainty, ensuring the algorithm never performs worse than anticipated.

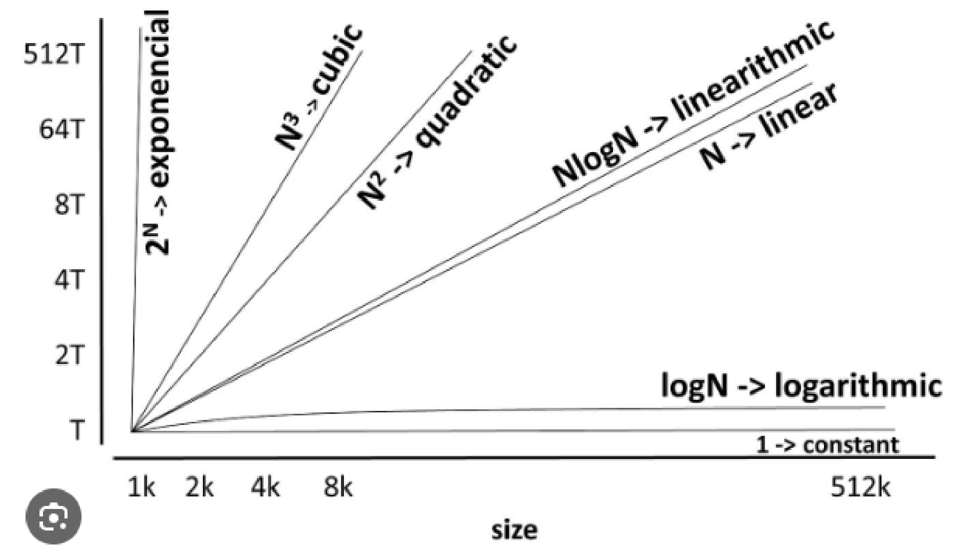
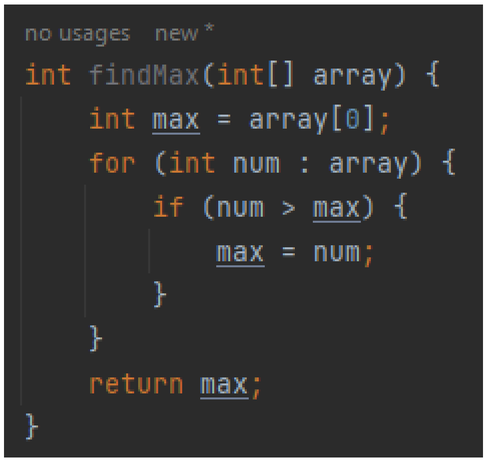


Figure 26: Big O Notation

1.3.4 Big O Example:



```
no usages   new *
int findMax(int[] array) {
    int max = array[0];
    for (int num : array) {
        if (num > max) {
            max = num;
        }
    }
    return max;
}
```

Figure 27: Big O Example Code

Explanation:

The purpose of this code is to find the maximum element in an array of integers.

int max = array[0];: Initialize the variable **max** with the first element of the array. This sets an initial value for comparison.

for (int num : array) {: This loop iterates through each element (num) in the given array.

if (num > max) { max = num; }: For each element in the array, the code compares it with the current value of max. If the current element (**num**) is greater than the current maximum (**max**), then the value of max is updated to the value of the current element (**num**).

return max;: After the loop completes, the maximum element found in the array is stored in the **max** variable, and this value is returned as the result.

Time Complexity Analysis (Big-O Notation):

In this code, there is a single loop that iterates through each element in the array exactly once. Inside the loop, there is a constant amount of work (comparison and assignment). Therefore, the time complexity of this algorithm can be described as **O(n),** where n is the number of elements in the array.

The reason it's **O(n)** is because the running time of the algorithm grows linearly with the input size (number of elements). As the input size doubles, the number of iterations in the loop also doubles, leading to a roughly proportional increase in running time.

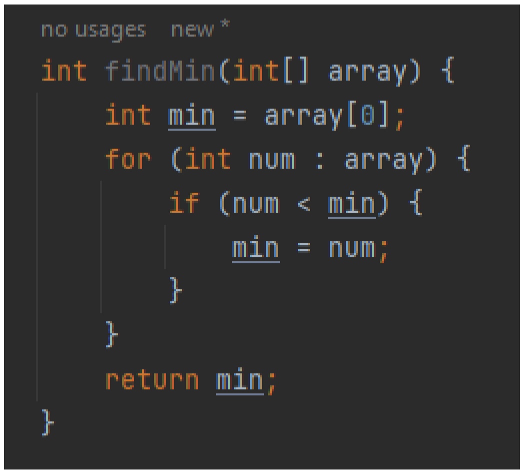
In summary, this code has a linear time complexity of **O(n),** making it efficient for finding the maximum element in an array.

1.3.5 Omega notation:

Omega notation (also known as Ω notation) is another notation used to describe the lower bound of an algorithm's performance. While Big-O notation provides an upper bound on the growth rate of an algorithm's running time or space usage, Omega notation provides a lower bound on the best-case scenario for an algorithm's performance.

In other words, if an algorithm has a lower bound of Ω(f(n)), it means that the algorithm will take at least as much time or space as f(n), as the input size grows.

Here's an example Java code snippet along with its Omega notation analysis:



```
no usages   new *
int findMin(int[] array) {
    int min = array[0];
    for (int num : array) {
        if (num < min) {
            min = num;
        }
    }
    return min;
}
```

Figure 28: Omega notation Example Code

Explanation:

This code is similar to the earlier example that found the maximum element, but it is designed to find the minimum element in an array of integers.

Time Complexity Analysis (Omega Notation):

For any non-empty input array, the algorithm will always perform at least one comparison during the loop, because it starts by initializing min with the first element of the array. This means that in the best-case scenario, the loop will execute at least once.

As a result, the best-case time complexity of this algorithm is Ω(1), because the algorithm always requires at least a constant amount of work, regardless of the input size.

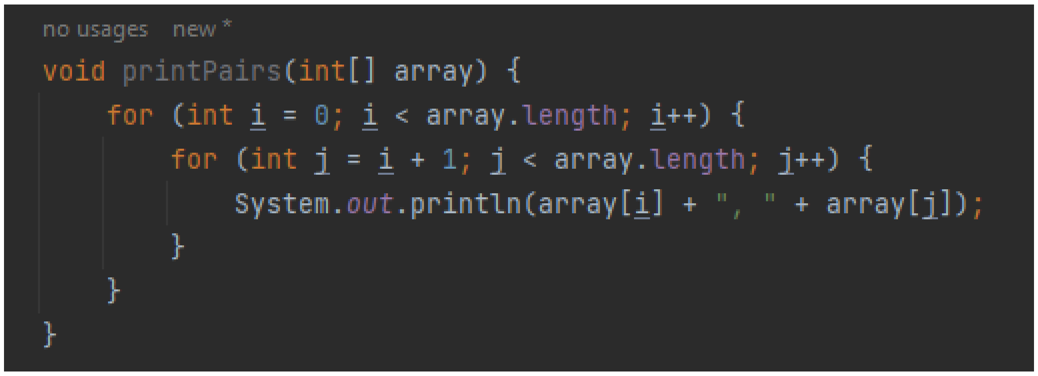
In summary, the Omega notation Ω(1) indicates that the algorithm's best-case time complexity is constant. This means that the algorithm is very efficient in terms of its best-case performance.

1.3.6 Theta Notation

Theta notation (θ notation) is a notation used to provide both the upper and lower bounds of an algorithm's performance. It gives a tighter bound on the growth rate of an algorithm's running time or space usage, showing that the algorithm's behavior is bounded both above and below by specific functions.

In other words, if an algorithm has a time complexity of θ(g(n)), it means that the algorithm's running time grows at the same rate as the function g(n) as the input size grows.

Here's an example Java code snippet along with its Theta notation analysis:



```
no usages   new *
void printPairs(int[] array) {
    for (int i = 0; i < array.length; i++) {
        for (int j = i + 1; j < array.length; j++) {
            System.out.println(array[i] + ", " + array[j]);
        }
    }
}
```

Figure 29: Theta notation

Explanation:

This code prints all possible pairs of elements from an input array of integers. It uses two nested loops to iterate through the array and print all combinations of pairs.

Time Complexity Analysis (Theta Notation):

The outer loop runs for **array.length** iterations. For each iteration of the outer loop, the inner loop runs for **array.length - i - 1** iterations. This results in the total number of pairs being printed to be approximately n * (n - 1) / 2, which is proportional to n^2.

Therefore, the time complexity of this algorithm is both an upper bound and a lower bound of θ(n^2). It's an upper bound because the algorithm cannot perform more work than proportional to n^2, and it's a lower bound because the algorithm's performance cannot be better than proportional to n^2.

In summary, the Theta notation θ(n^2) indicates that the algorithm's time complexity grows quadratically with the input size. It captures both the best-case and worst-case scenarios for this algorithm.

1.4 Determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example. (P7)

1.4.1 Time Complexity

A, What is Time complexity

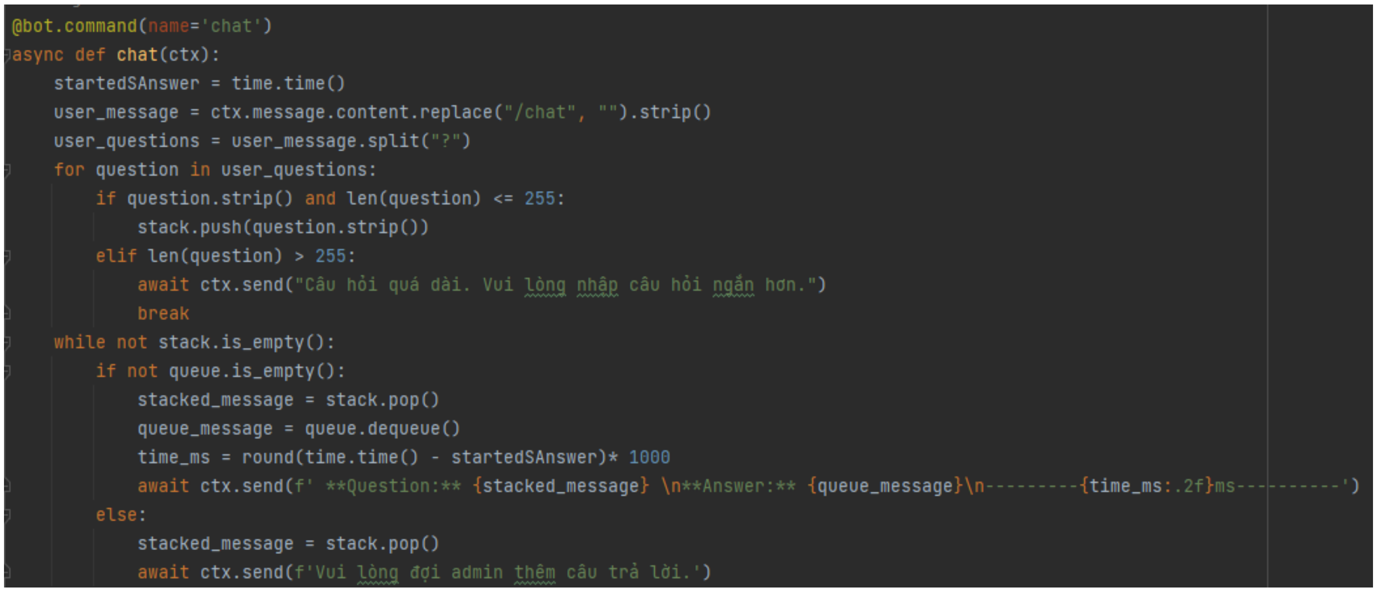
Time complexity refers to the duration an algorithm requires to execute, expressed as a function of the input size. It quantifies the time taken for each code statement in the algorithm to run. Rather than indicating the overall execution time, it offers insights into how the execution time changes (either grows or diminishes) in relation to the number of operations (increasing or decreasing) within the algorithm.

The provided Big O chart illustrates that the most favorable scenario is O(1), denoting constant time complexity. This signifies that your algorithm carries out a single operation without any need for iteration. Following that is O(log n), which is also favorable, along with similar complexities, as depicted in the chart below:

- O(1) - Excellent/Best
- O(log n) - Good
- O(n) - Fair
- O(n log n) - Bad
- O(n^2), O(2^n) and O(n!) - Horrible/Worst

B, Example

Consider the following Python code that processes user questions and provides answers. We'll analyze its time complexity.



```python
@bot.command(name='chat')
async def chat(ctx):
    startedSAnswer = time.time()
    user_message = ctx.message.content.replace("/chat", "").strip()
    user_questions = user_message.split("?")
    for question in user_questions:
        if question.strip() and len(question) <= 255:
            stack.push(question.strip())
        elif len(question) > 255:
            await ctx.send("Câu hỏi quá dài. Vui lòng nhập câu hỏi ngắn hơn.")
            break
    while not stack.is_empty():
        if not queue.is_empty():
            stacked_message = stack.pop()
            queue_message = queue.dequeue()
            time_ms = round(time.time() - startedSAnswer)* 1000
            await ctx.send(f' **Question:** {stacked_message} \n**Answer:** {queue_message}\n---------{time_ms:.2f}ms----------')
        else:
            stacked_message = stack.pop()
            await ctx.send(f'Vui lòng đợi admin thêm câu trả lời.')
```

Figure 30: Time Complexity in my code

In this code, the time complexity is influenced by two primary loops:

The first loop iterates through user_questions, which has a time complexity of O(n), where n is the number of questions.

The second loop processes elements in the stack and queue. Since these operations are dependent on the number of questions, the overall time complexity remains O(n).

Therefore, the time complexity of this code is O(n), where n is the number of questions.