PROBLEM 6 REPORT

FILE ALLOCATION STRATEGIES A.) SEQUENTIAL B.) LINKED C.) INDEXED

GLORY DEJI-AKINLOTAN- IT210AL

INTRODUCTION

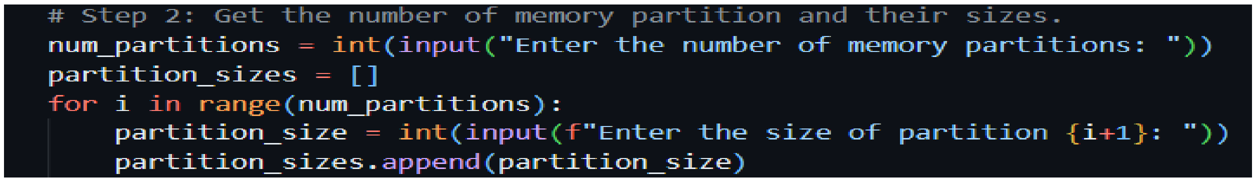
- The object of problem 6 is to implement the concepts of file allocation strategies. This report discusses the implementation of file allocation strategies - Sequential, Indexed, and Linked. The objective of this report is to explain the algorithms used for each of these techniques and their corresponding outputs. The report provides a detailed description of each algorithm and images of the code are in the report as well. The sequential file allocation strategy follows a first-fit, best-fit, or worst-fit algorithm to allocate memory blocks for processes. The Indexed file allocation strategy checks for buffer capacity and waits for producer and consumer. The linked file allocation strategy uses a queue to hold all pages in memory and a stack to replace pages when a page fault occurs. The report includes the algorithms for all three file allocation strategies and the corresponding outputs for each of the algorithms.

SEQUENTIAL FILE ALLOCATION

This Python code simulates three different memory management algorithms - First Fit, Best Fit, and Worst Fit - to allocate memory to a number of processes. Here's a report based on the code:

- **Step 1: Start the program**. The program starts by asking the user to input the number of memory partitions and their sizes, and the number of processes and their sizes.
- **Step 2: Get the number of memory partition and their sizes**. The user is prompted to enter the number of memory partitions and their sizes using a for loop. The code then appends the partition sizes to the **partition_sizes** list for each iteration. This list is used later in the program to allocate memory to the processes using different memory allocation algorithms.

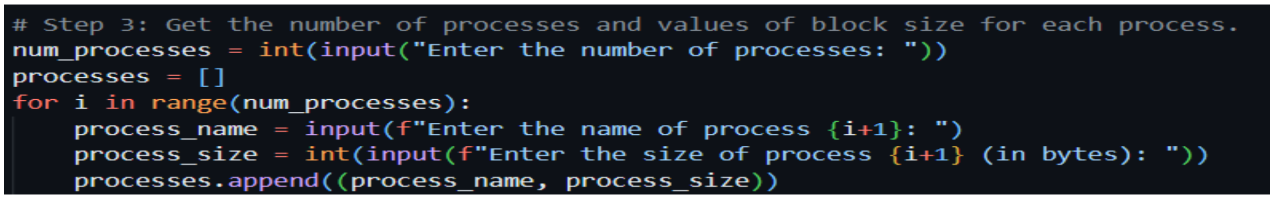
The **partition_sizes** list is used to keep track of the size of each memory partition. The index of each partition in the list corresponds to the partition number, so the size of the partition at index (i) can be accessed using **partition_sizes[i]**.



```
# Step 2: Get the number of memory partition and their sizes.
num_partitions = int(input("Enter the number of memory partitions: "))
partition_sizes = []
for i in range(num_partitions):
    partition_size = int(input(f"Enter the size of partition {i+1}: "))
    partition_sizes.append(partition_size)
```

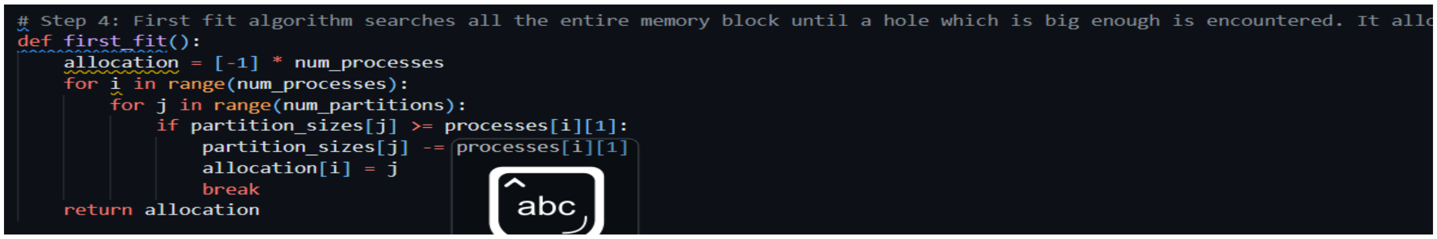
- **Step 3: Get the number of processes and values of block size for each process.** The user is prompted to enter the number of processes and the size of each process in bytes using another for a loop. It does this using a for loop which iterates over the range of the number of processes entered, and for each iteration, it prompts the user to enter the name and size of the process. The process name is input using the **input()** function and the process size is input as an integer using the **int(input())** function.

- The code then adds a tuple containing the process name and size to the **processes** list for each iteration. This list is used later in the program to allocate memory to the processes using different memory allocation algorithms.



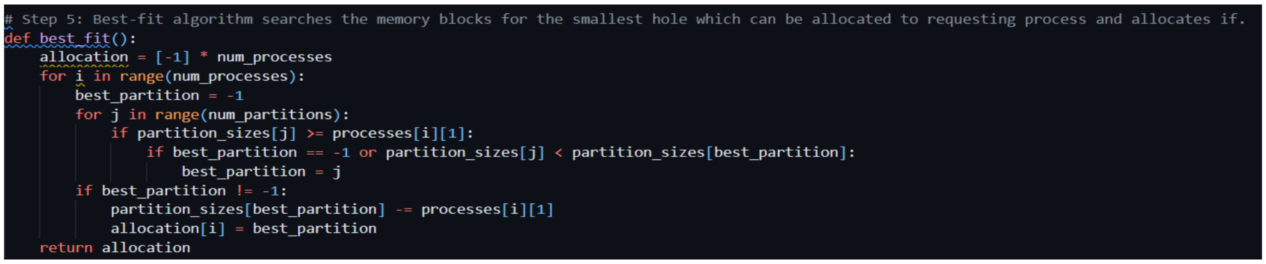
```
# Step 3: Get the number of processes and values of block size for each process.
num_processes = int(input("Enter the number of processes: "))
processes = []
for i in range(num_processes):
    process_name = input(f"Enter the name of process {i+1}: ")
    process_size = int(input(f"Enter the size of process {i+1} (in bytes): "))
    processes.append((process_name, process_size))
```

- **Step 4: First fit algorithm searches all the entire memory block until a hole which is big enough is encountered. It allocates that memory block for the requesting process**. The first fit algorithm is implemented using a nested for loop. It searches for the first partition that is big enough to accommodate the process and allocates it to that partition. The **first_fit** function implements the first fit algorithm for memory allocation. It iterates over each process and memory partition, and if it finds a partition with sufficient space for a process, it allocates that partition to the process and moves to the next process.



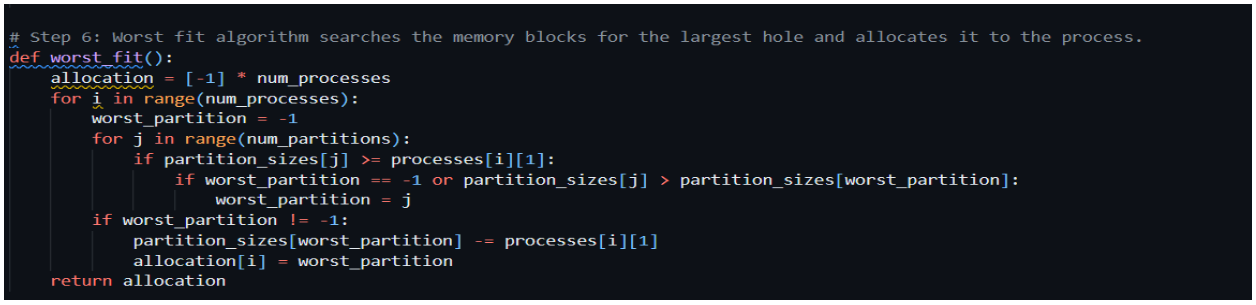
```
# Step 4: First fit algorithm searches all the entire memory block until a hole which is big enough is encountered. It allo
def first_fit():
    allocation = [-1] * num_processes
    for i in range(num_processes):
        for j in range(num_partitions):
            if partition_sizes[j] >= processes[i][1]:
                partition_sizes[j] -= processes[i][1]
                allocation[i] = j
                break
    return allocation
```

- **Step 5: Best-fit algorithm searches the memory blocks for the smallest hole which can be allocated to requesting process and allocates if**. The best-fit algorithm is implemented using a nested for loop. It searches for the smallest partition that can accommodate the process and allocates it to that partition.



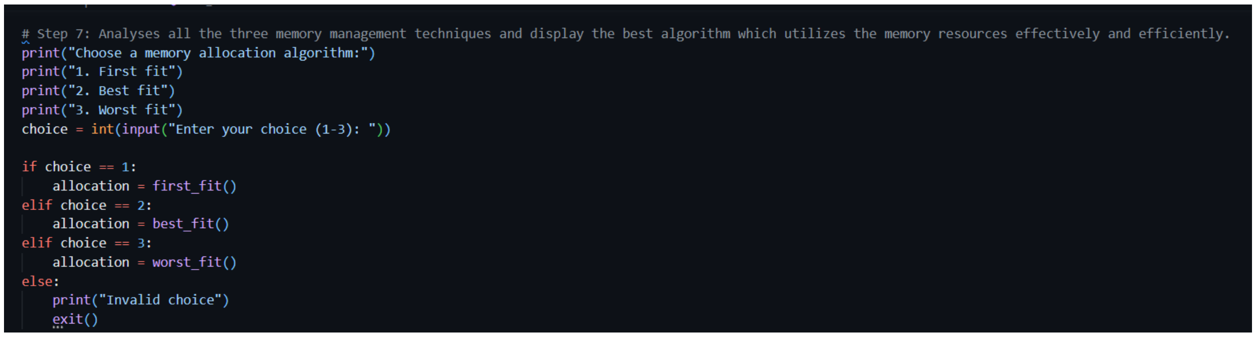
```
# Step 5: Best-fit algorithm searches the memory blocks for the smallest hole which can be allocated to requesting process and allocates if.
def best_fit():
    allocation = [-1] * num_processes
    for i in range(num_processes):
        best_partition = -1
        for j in range(num_partitions):
            if partition_sizes[j] >= processes[i][1]:
                if best_partition == -1 or partition_sizes[j] < partition_sizes[best_partition]:
                    best_partition = j
        if best_partition != -1:
            partition_sizes[best_partition] -= processes[i][1]
            allocation[i] = best_partition
    return allocation
```

- **Step 6: Worst fit algorithm searches the memory blocks for the largest hole and allocates it to the process**. The worst-fit algorithm is implemented using a nested for loop. It searches for the largest partition that can accommodate the process and allocates it to that partition.



```python
# Step 6: Worst fit algorithm searches the memory blocks for the largest hole and allocates it to the process.
def worst_fit():
    allocation = [-1] * num_processes
    for i in range(num_processes):
        worst_partition = -1
        for j in range(num_partitions):
            if partition_sizes[j] >= processes[i][1]:
                if worst_partition == -1 or partition_sizes[j] > partition_sizes[worst_partition]:
                    worst_partition = j
        if worst_partition != -1:
            partition_sizes[worst_partition] -= processes[i][1]
            allocation[i] = worst_partition
    return allocation
```

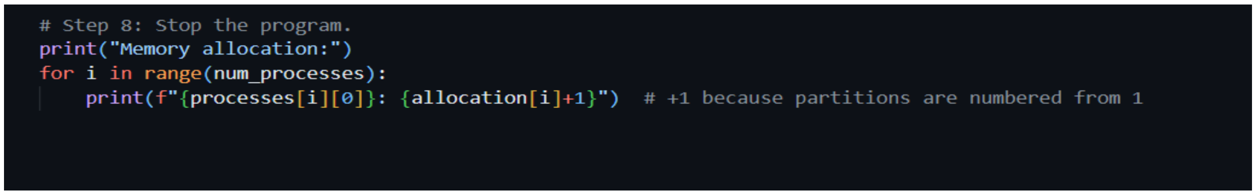
- **Step 7: Analyses all three memory management techniques and display the best algorithm which utilizes the memory resources effectively and efficiently**. The user is prompted to choose which memory allocation algorithm to use - first fit, best fit, or worst fit - and the program runs the corresponding function. The output shows which partition each process is allocated to.



```python
# Step 7: Analyses all the three memory management techniques and display the best algorithm which utilizes the memory resources effectively and efficiently.
print("Choose a memory allocation algorithm:")
print("1. First fit")
print("2. Best fit")
print("3. Worst fit")
choice = int(input("Enter your choice (1-3): "))

if choice == 1:
    allocation = first_fit()
elif choice == 2:
    allocation = best_fit()
elif choice == 3:
    allocation = worst_fit()
else:
    print("Invalid choice")
    exit()
```

- **Step 8: Stop the program.** The program ends by printing the allocation results to the console.

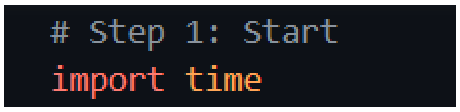


```python
# Step 8: Stop the program.
print("Memory allocation:")
for i in range(num_processes):
    print(f"{processes[i][0]}: {allocation[i]+1}")  # +1 because partitions are numbered from 1
```

Overall, this code is a simple implementation of three common memory management algorithms. The program provides a way to compare the effectiveness and efficiency of each algorithm in allocating memory resources to a number of processes.

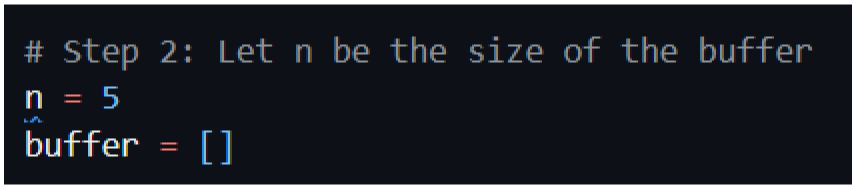
INDEXED FILE ALLOCATION

- **Step 1: start.** This imports the ‘time’ module for use later in the program.



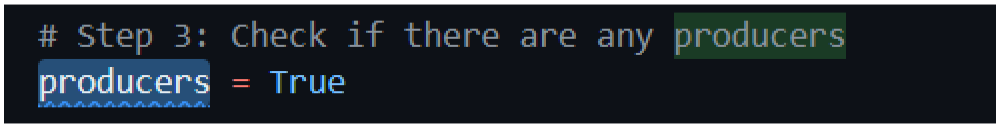
```python
# Step 1: Start
import time
```

- **Step 2: Let n be the size of the buffer.** This sets the size of the buffer to 5 and initializes an empty list to serve as the buffer.



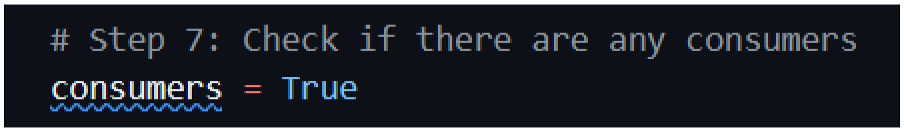
```python
# Step 2: Let n be the size of the buffer
n = 5
buffer = []
```

- **Step 3: Check if there are any producers.** This step initializes a boolean variable producer to True.



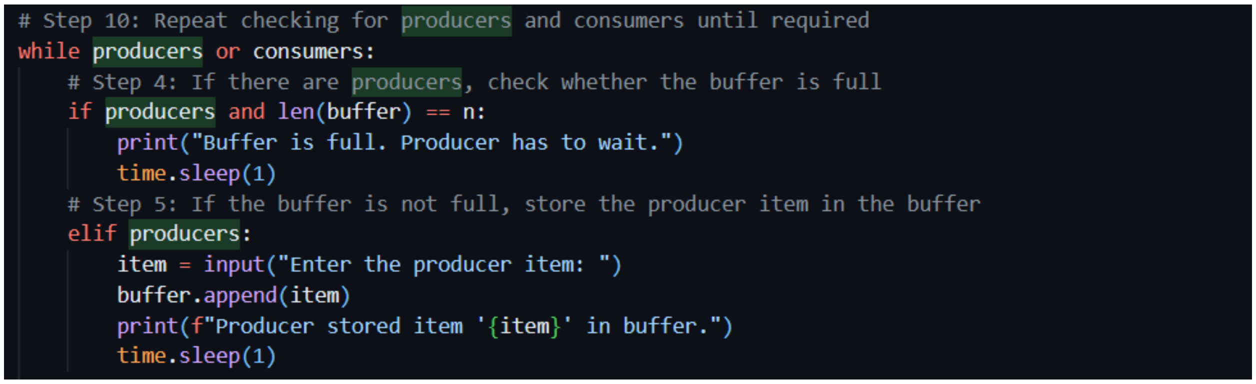
```python
# Step 3: Check if there are any producers
producers = True
```

- **Step 7: Check if there are any consumers.** This initializes a boolean variable consumers to True.



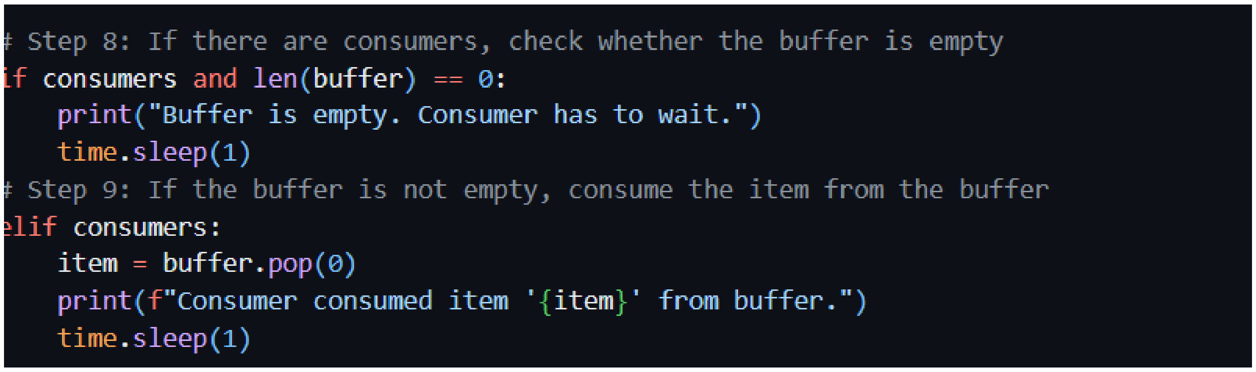
```python
# Step 7: Check if there are any consumers
consumers = True
```

- **Step 10: Repeat checking for producers and consumers until required.** This sets up a loop that will continue until both producers and consumers are False.
- **Step 4: If there are producers, check whether the buffer is full.** This checks if there are any producers and if the buffer is full. If so, it prints a message and waits for 1 second using the time.sleep() function.
- **Step 5: If the buffer is not full, store the producer item in the buffer.** This checks if there are any producers and if the buffer is not full. If so, it prompts the user to enter a producer item, adds it to the buffer, prints a message and waits for 1 second.



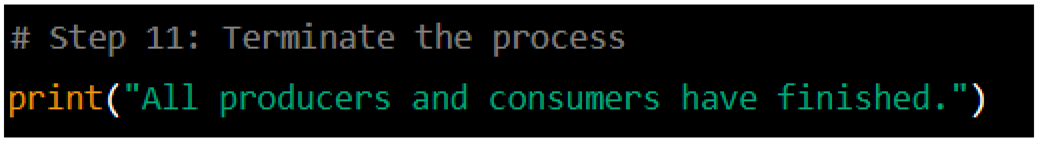
```python
# Step 10: Repeat checking for producers and consumers until required
while producers or consumers:
    # Step 4: If there are producers, check whether the buffer is full
    if producers and len(buffer) == n:
        print("Buffer is full. Producer has to wait.")
        time.sleep(1)
    # Step 5: If the buffer is not full, store the producer item in the buffer
    elif producers:
        item = input("Enter the producer item: ")
        buffer.append(item)
        print(f"Producer stored item '{item}' in buffer.")
        time.sleep(1)
```

- **Step 8: If there are consumers, check whether the buffer is empty.** This checks if there are any consumers and if the buffer is empty. If so, it prints a message and waits for 1 second.
- **Step 9: If the buffer is not empty, consume the item from the buffer**. This checks if there are any consumers and if the buffer is not empty. If so, it removes the first item from the buffer, prints a message and waits for 1 second.



```
# Step 8: If there are consumers, check whether the buffer is empty
if consumers and len(buffer) == 0:
    print("Buffer is empty. Consumer has to wait.")
    time.sleep(1)
# Step 9: If the buffer is not empty, consume the item from the buffer
elif consumers:
    item = buffer.pop(0)
    print(f"Consumer consumed item '{item}' from buffer.")
    time.sleep(1)
```

- **Step 11: Terminate the process**. This prints a message indicating that all producers and consumers have finished, and the program terminates.



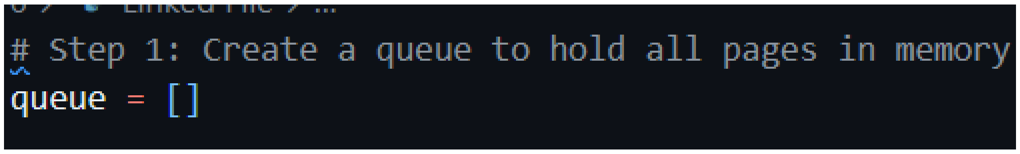
```
# Step 11: Terminate the process
print("All producers and consumers have finished.")
```

- ***Steps 3 and 7: Check if there are any producers or consumers left.*** This prompts the user to enter whether there are any more producers and consumers. It then sets the producers and consumers variables based on the user input.

❖ This code is an implementation of a producer-consumer problem where producers and consumers share a common buffer. The program starts by initializing a buffer of size 5 and checks if there are any producers or consumers. It then repeatedly checks whether the buffer is full or empty and allows producers to store items in the buffer if it's not full and allows consumers to consume items from the buffer if it's not empty. The program terminates when there are no more producers or consumers.

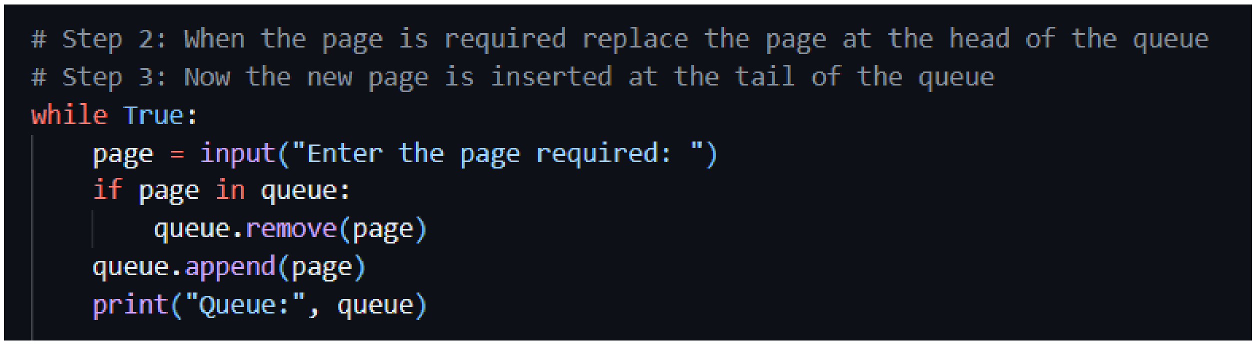
<u>LINKED FILE ALLOCATION</u>

- **Step 1: Create a queue to hold all pages in memory.** This line initializes an empty list named queue, which will be used to store pages.



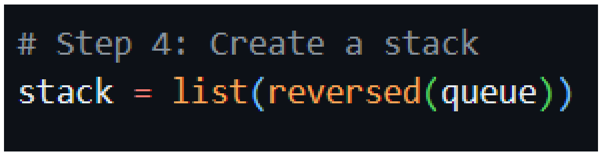
```
# Step 1: Create a queue to hold all pages in memory
queue = []
```

- **Step 2: When the page is required replace the page at the head of the queue.** This while loop keeps running until the user enters the "stop" command. Each iteration of the loop prompts the user to enter a page that is required. If the page is already in the queue, it is removed so that it can be added to the tail of the queue.
- **Step 3: Now the new page is inserted at the tail of the queue.** This line prints the updated queue after each page is added.



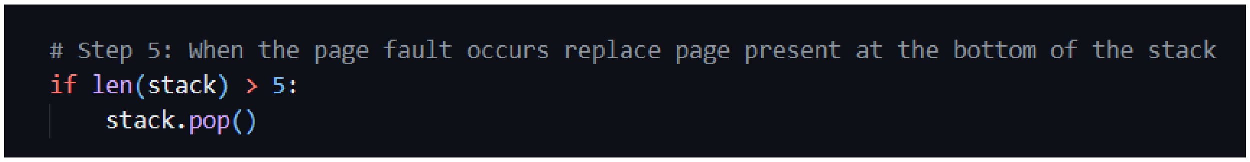
```python
# Step 2: When the page is required replace the page at the head of the queue
# Step 3: Now the new page is inserted at the tail of the queue
while True:
    page = input("Enter the page required: ")
    if page in queue:
        queue.remove(page)
    queue.append(page)
    print("Queue:", queue)
```

- **Step 4: Create a stack**. This line creates a new list named **stack** by reversing the order of elements in the **queue** list. Meaning that the most recently used page is at the top of the stack, while the least recently used page is at the bottom.



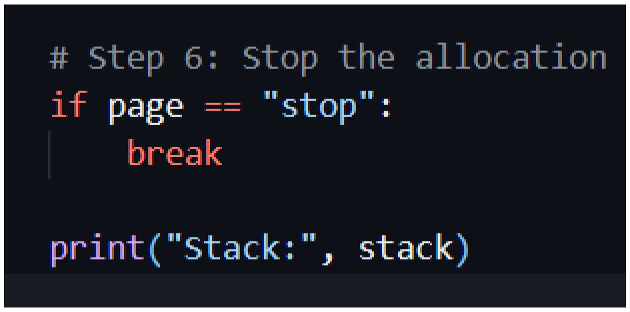
```python
# Step 4: Create a stack
stack = list(reversed(queue))
```

- **Step 5: When the page fault occurs replace page present at the bottom of the stack.** This if statement checks if the stack list has more than 5 elements. If so, it removes the bottom element of the stack using the pop() method.



```python
# Step 5: When the page fault occurs replace page present at the bottom of the stack
if len(stack) > 5:
    stack.pop()
```

- **Step 6: Stop the allocation.** This if statement checks if the user has entered "stop", and if so, it breaks out of the while loop**.**



```python
# Step 6: Stop the allocation
if page == "stop":
    break

print("Stack:", stack)
```

- The last line prints the updated stack after each page is added or removed, so you can see which pages are currently in memory and how recently they were used.

❖ This code implements the page replacement algorithm for managing memory pages. It uses a queue data structure to hold all pages currently in memory, where the page at the head of the queue is the oldest one. When a new page is required, the algorithm removes the oldest page from the head of the queue and inserts the new page at the tail of the queue. It then creates a stack of pages in the reversed order of the queue, where the page at the bottom of the stack is the oldest one. If the length of the stack exceeds a certain limit (in this case, 5), the algorithm removes the oldest page from the bottom of the stack. This process is repeated until the user enters "stop" to terminate the algorithm.