1/1/2024

Queue implementation using array

Converting it by MIPS

Subjected to   
Prof/ Lamia Alrefaai

[Introduction to Queues 1](#_Toc165727377)

[Types 1](#_Toc165727378)

[Operations 1](#_Toc165727379)

[Applications 1](#_Toc165727380)

[Circular Queue 2](#_Toc165727381)

[Definition 2](#_Toc165727382)

[Implementation 2](#_Toc165727383)

[Input 2](#_Toc165727384)

[Working algorithm 3](#_Toc165727385)

[Output 4](#_Toc165727386)

[Register values & values in memory segments 4](#_Toc165727387)

[Code 5](#_Toc165727388)

[Java code 5](#_Toc165727389)

[Assembly code 9](#_Toc165727390)

[Priority Queue 14](#_Toc165727391)

[Definition 14](#_Toc165727392)

[Implementation 14](#_Toc165727393)

[Working algorithm 14](#_Toc165727394)

[Input 15](#_Toc165727395)

[Output 15](#_Toc165727396)

[Register values & values in memory segments 16](#_Toc165727397)

[Code: 17](#_Toc165727398)

[Java code 17](#_Toc165727399)

[Assembly code 20](#_Toc165727400)

[Life application: Bank Teller Services 26](#_Toc165727401)

[Application Statement 26](#_Toc165727402)

[Implementation 26](#_Toc165727403)

[Working algorithm 26](#_Toc165727404)

[Input 28](#_Toc165727405)

[Output 28](#_Toc165727406)

[Register values & values in memory segments 29](#_Toc165727407)

[Code 30](#_Toc165727408)

[Java code 30](#_Toc165727409)

[Assembly c ode 32](#_Toc165727410)

[Task Management 35](#_Toc165727411)

# 

# Introduction to Queues

Queues are a fundamental concept in computer science, providing a structured approach to managing data. Unlike other data structures, queues enforce the First-In-First-Out (FIFO) principle, meaning the first element added is the first to be removed. This principle mirrors real-world scenarios like waiting in line, ensuring fairness and order. Throughout this report, we'll delve into the mechanics of queues, exploring their operations and applications, and highlighting their importance in various computing contexts.

## Types

1. **Linear Queue:** Simple and straightforward, best for scenarios where processing order strictly follows arrival order.
2. **Circular Queue:** Efficient use of memory with a fixed-size structure, suitable for scenarios where elements are continuously added and removed.
3. **Priority Queue:** Ideal for scenarios where processing order is based on priority levels rather than arrival order, allowing for efficient handling of high-priority tasks.

## Operations

1. **Enqueue (Insert):** Add an element to the rear end of the queue.
2. **Dequeue (Remove):** Remove and retrieve the element from the front end of the queue.
3. **IsEmpty:** Check if the queue is empty.
4. **IsFull:** Check if the queue is full (applies to fixed-size queues).
5. **Peek Front:** Retrieve the element at the front end of the queue without removing it.
6. **Peek Rear:** Retrieve the element at the rear end of the queue without removing it.
7. **Size:** Get the number of elements currently in the queue.

## Applications

* Operating Systems
* Networking
* Data Structures
* Multithreading and Parallel Computing
* Simulation and Modeling
* Print Spooling

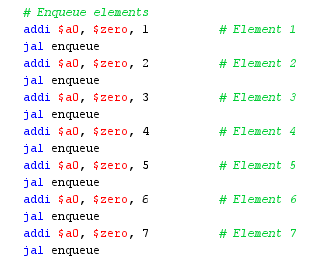
# Circular Queue

## Definition

A data structure that simulates a queue with a fixed-size buffer is called a circular queue. When the rear pointer of a circular queue hits the end of the underlying array, it wraps around to the beginning of the array, creating a circular arrangement as opposed to a linear queue, where it terminates at that point. When a fixed-size queue is required, this approach makes effective use of memory because elements can be added and removed continually without taking up space. The element that is enqueued first will be dequeued first in a circular queue, which adheres to the First In, First Out (FIFO) concept.

## Implementation

### Input



### Working algorithm

1. **Initialize the variables:**

* Load the base address of the items array into register $t0.
* Load the capacity of the queue into register $t1.
* Load the number of items in the queue into register $t2.
* Load the constant 1 into register $t3.

1. **Enqueue elements:**

* Check if the queue is full by comparing $t2 (number of items) with $t1 (capacity).
* If the queue is not full:
* Calculate the offset for the next available slot in the items array.
* Calculate the address of the next slot using the base address and offset.
* Store the element to be enqueued in the next slot.
* Increment the number of items.
* Print "Inserted: " followed by the inserted element.
* If the queue is full, print "Queue is full!".

1. **Dequeue elements:**

* Load the base address of the items array into register $s5.
* Load the element at the front of the queue into register $t7.
* Check if the queue is empty by comparing $t2 (number of items) with zero.
* If the queue is not empty:
* Shift the elements in the items array to the left to remove the front element.
* Decrement the number of items.
* Print "Deleted: " followed by the dequeued element.
* If the queue is empty, print "Queue is empty!".

1. **Size of the queue:**

* Print "Size: " followed by the number of items in the queue.

1. **Peek at the front element:**

* Check if the queue is empty.
* If the queue is not empty, print "peekFront: " followed by the front element.

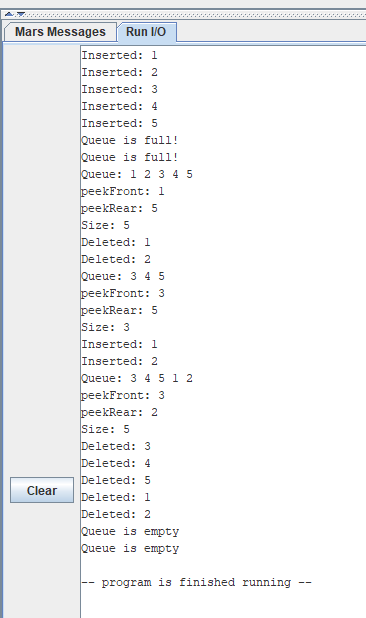
1. **Peek at the rear element:**

* Check if the queue is empty.
* If the queue is not empty:
* Calculate the address of the last element in the items array.
* Print "peekRear: " followed by the last element.

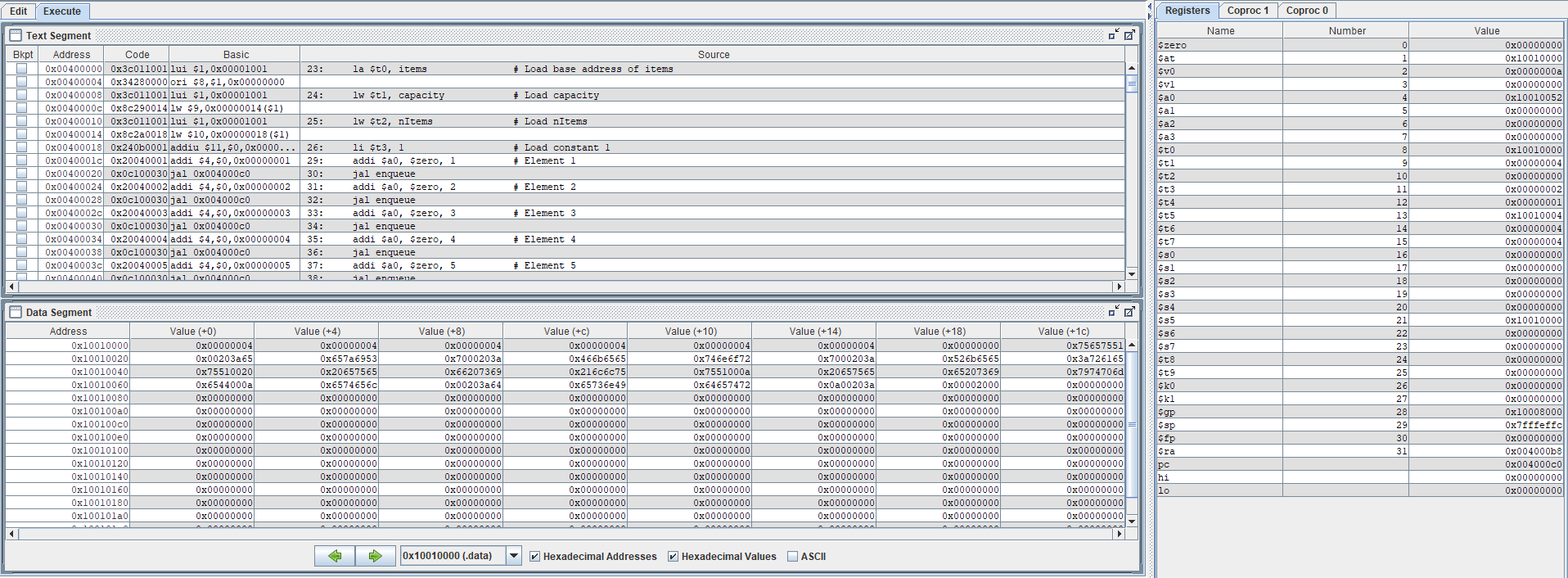
1. **Display the queue:**

* Check if the queue is empty.
* If the queue is not empty:
* Print "Queue: ".
* Iterate over the items array and print each element followed by a space.

### Output



### Register values & values in memory segments



## Code

### Java code



A screen shot of a computer program

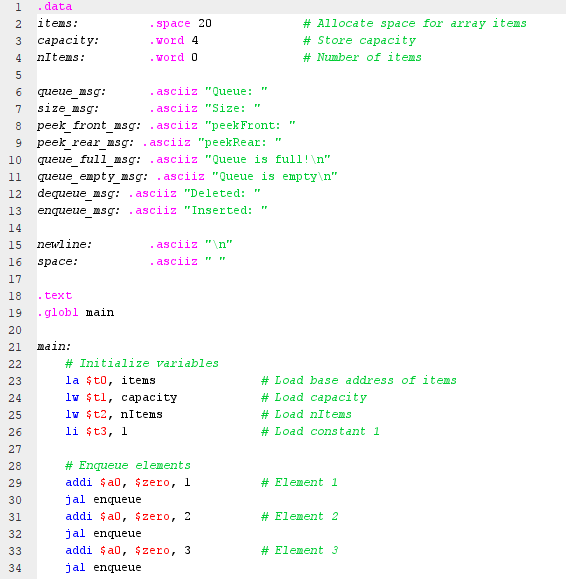
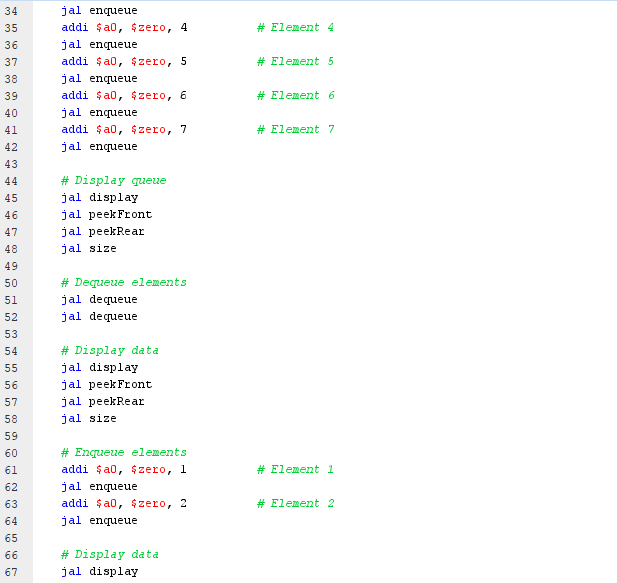
Description automatically generatedA computer screen shot of code

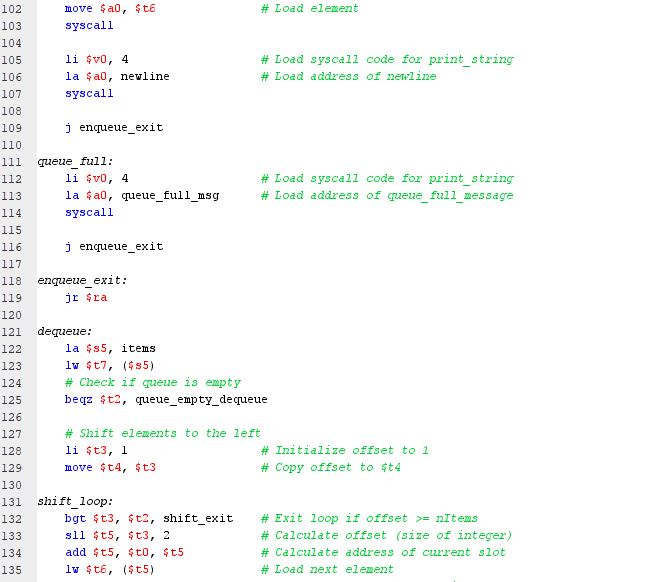
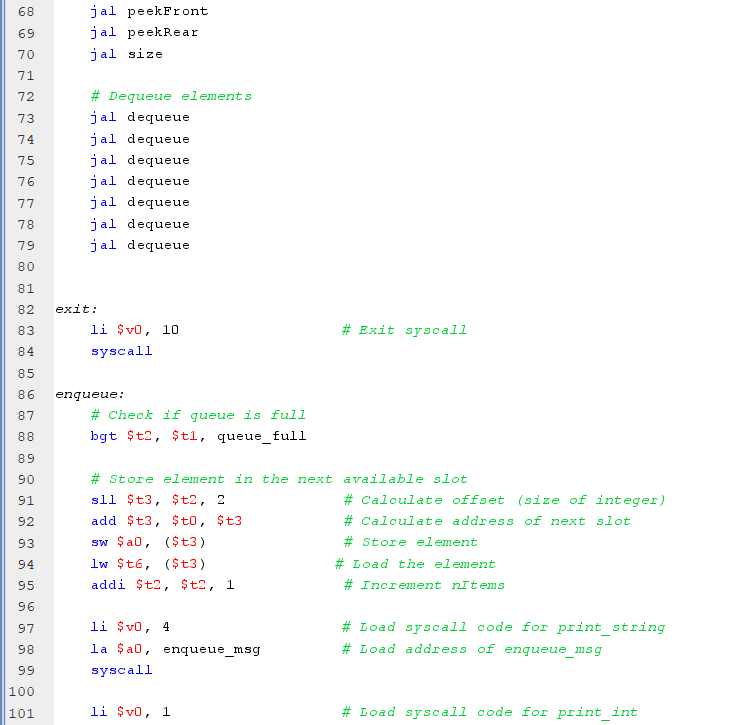
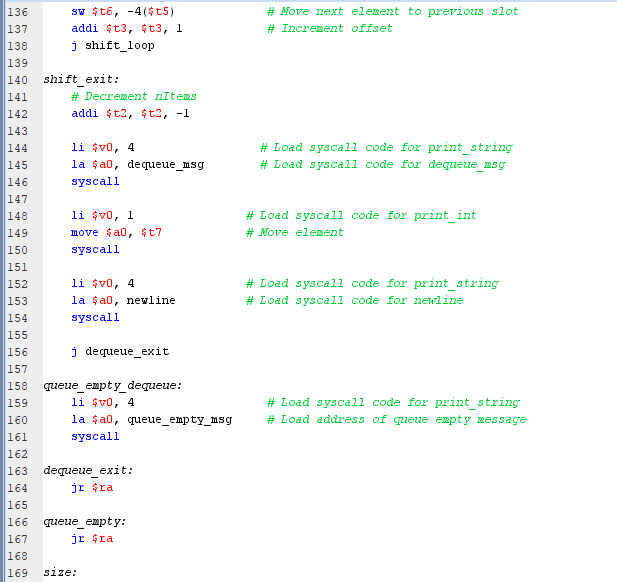
Description automatically generatedA screen shot of a computer

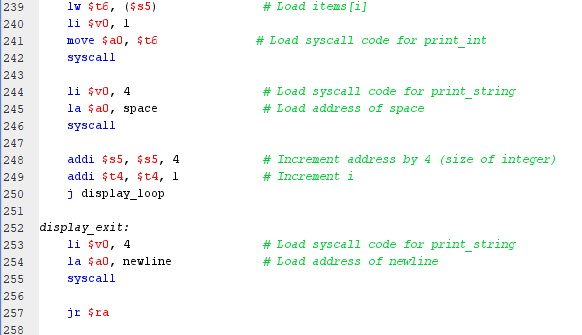
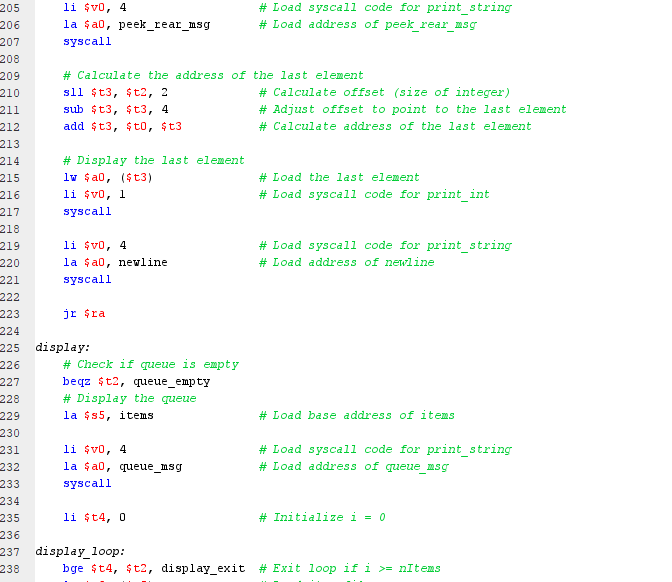
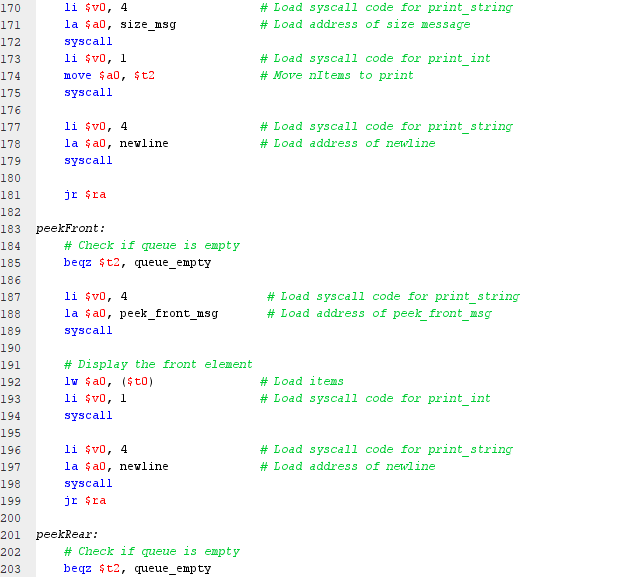
Description automatically generatedA screen shot of a computer code

Description automatically generated

### Assembly code

****



# 

# Priority Queue

## Definition

A priority queue is a data structure that manages elements based on their priority levels, ensuring that higher priority elements are dequeued before lower priority ones. Unlike a traditional queue where elements are retrieved in the order they were added, a priority queue allows for efficient retrieval of the most important element. This data structure finds applications in scenarios where tasks or events need to be processed based on their urgency or importance, such as task scheduling in operating systems or network packet scheduling. Priority queues can be implemented using various underlying data structures like binary heaps, priority heaps, or balanced binary search trees, each offering different performance characteristics to suit specific requirements.

## Implementation

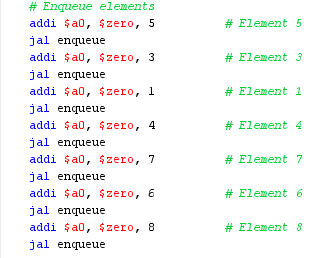
### Working algorithm

The provided MIPS assembly code appears to be implementing a priority queue data structure. It initializes some variables, enqueues elements into the queue, dequeues elements, displays the queue, and performs other operations like finding the maximum and minimum elements.

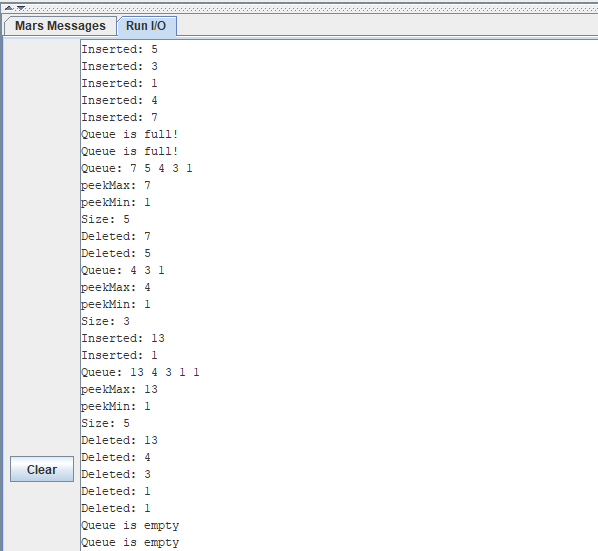
Here's a brief overview of the code's functionality:

1. **Enqueue (enqueue):**
   * Inserts elements into the queue while maintaining the priority order. It finds the correct position for insertion, shifts elements to make space, and then inserts the new element.
2. **Dequeue (dequeue):**
   * Removes the front element from the queue by shifting the remaining elements to the left.
3. **Display (display):**
   * Prints the elements in the queue.
4. **Size (size):**
   * Prints the current size of the queue.
5. **PeekMax (peekMax):**
   * Prints the maximum element in the queue.
6. **PeekMin (peekMin):**
   * Prints the minimum element in the queue.
7. **Sort (sort):** 
   * This part of the code seems to be an attempt to implement a sorting algorithm, possibly bubble sort or a similar sorting algorithm. However, the implementation seems incomplete, and there are some errors in the loop conditions and variable usage.

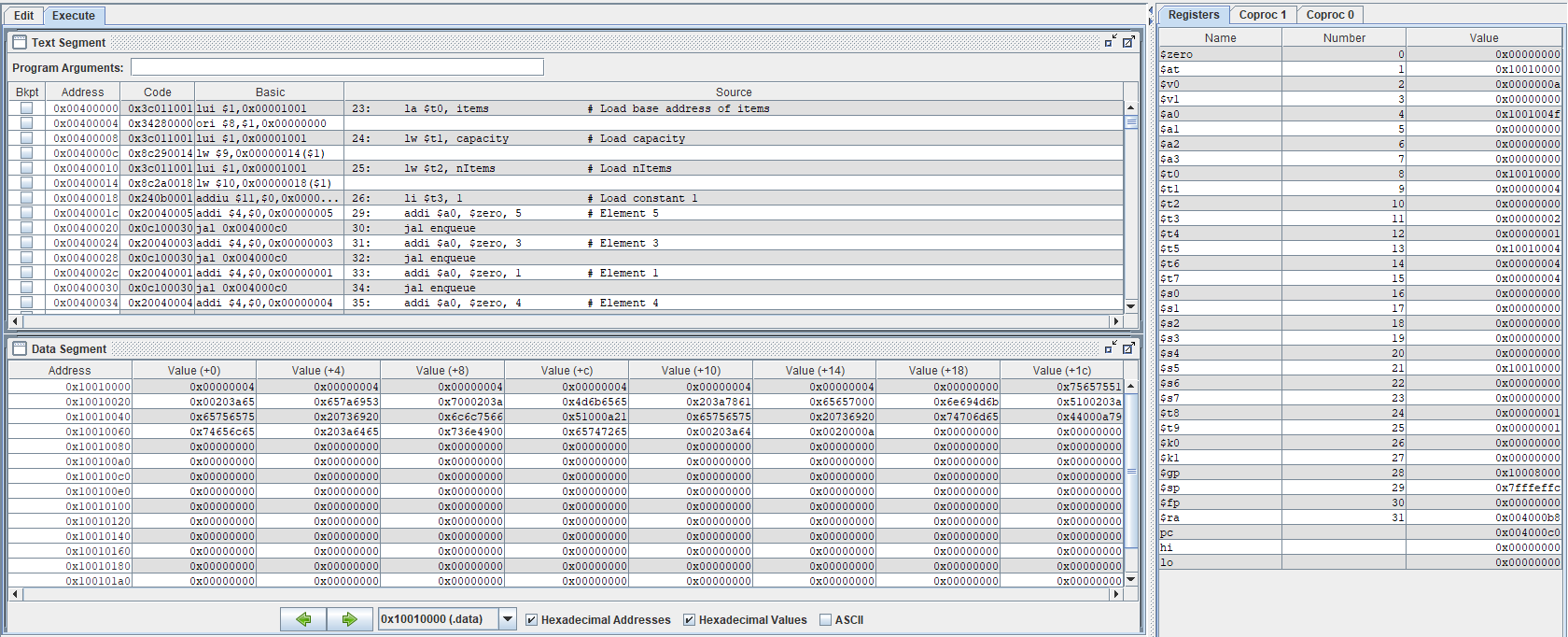
### Input



### Output

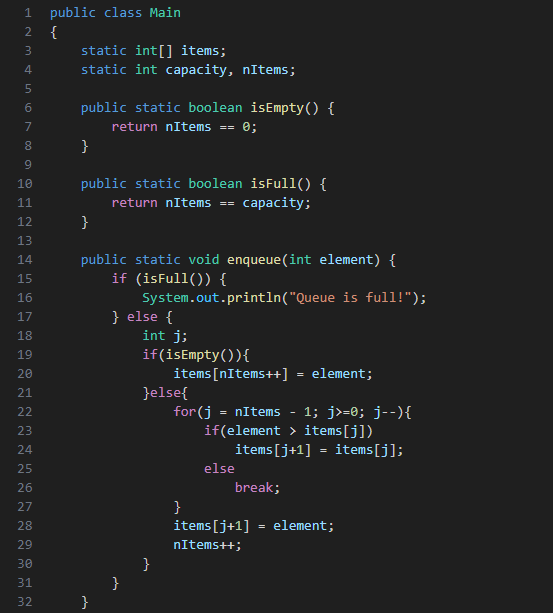


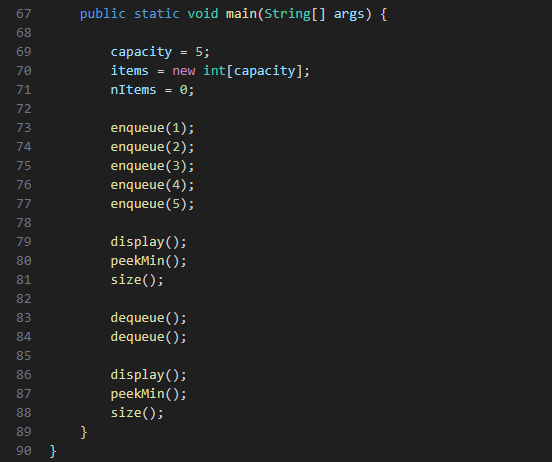
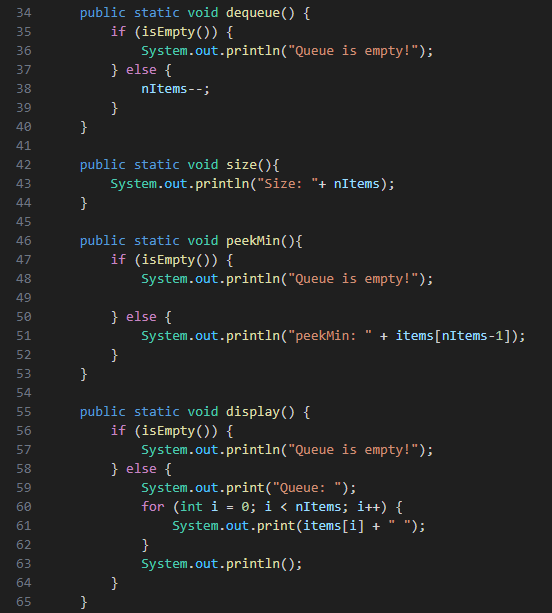
### Register values & values in memory segments



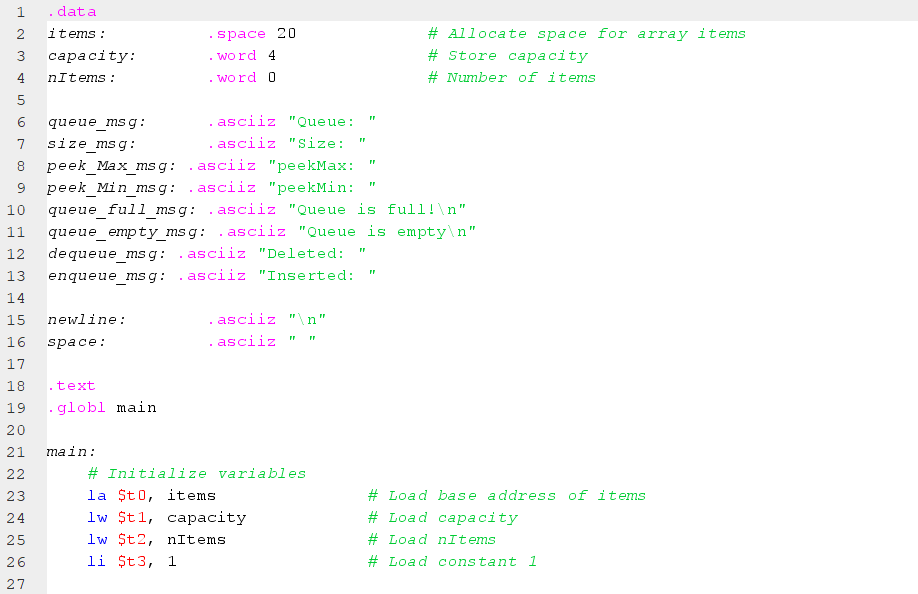
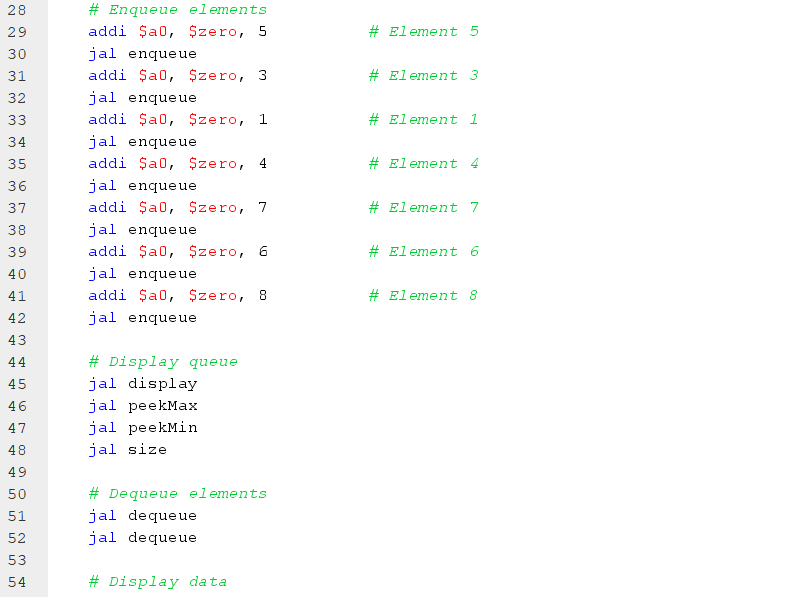
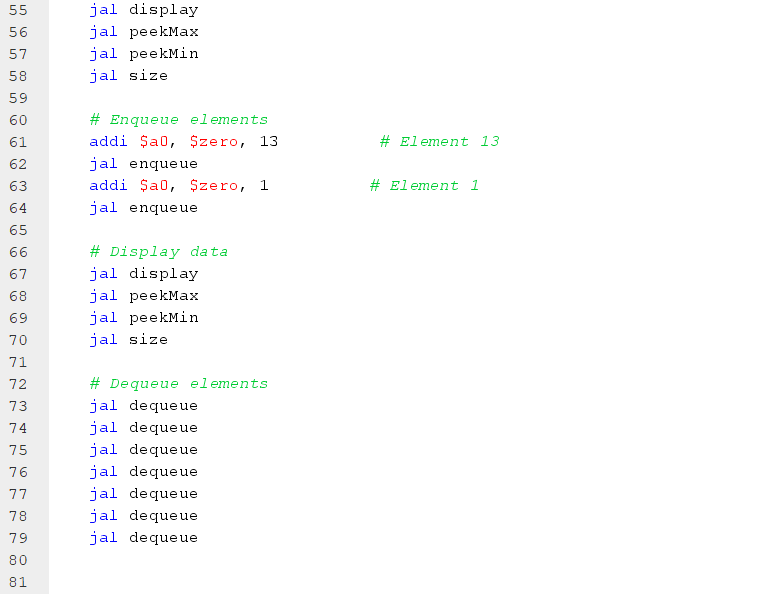
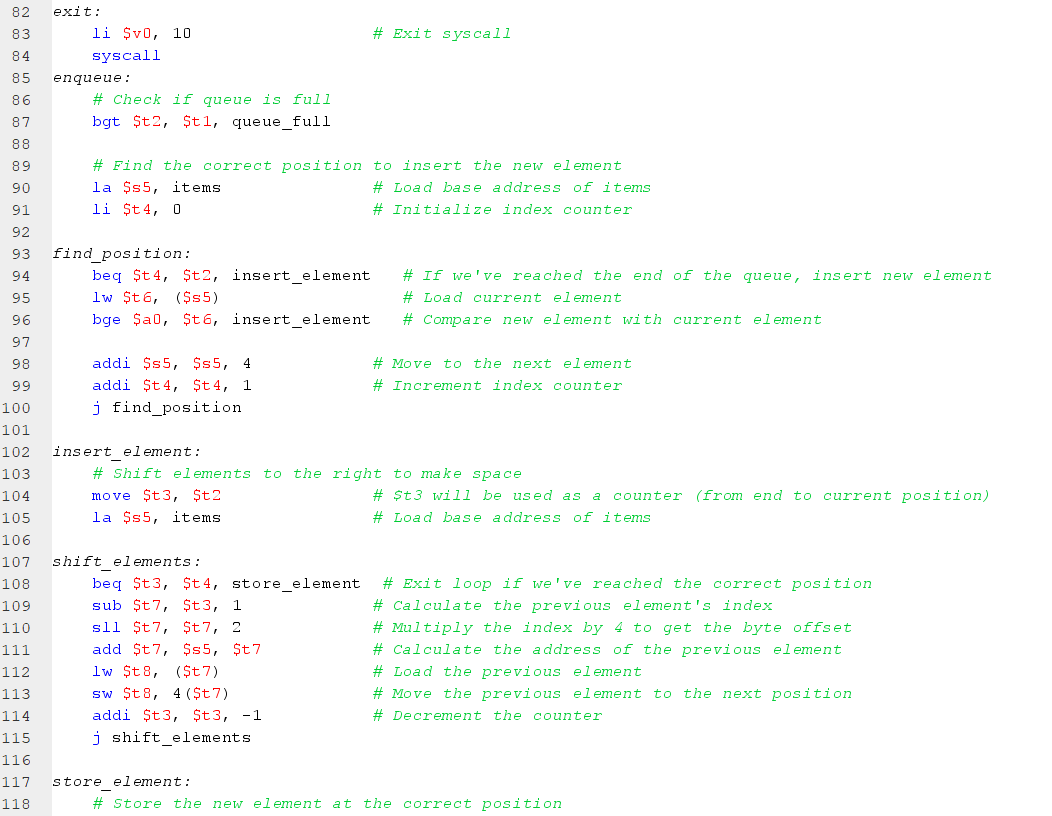
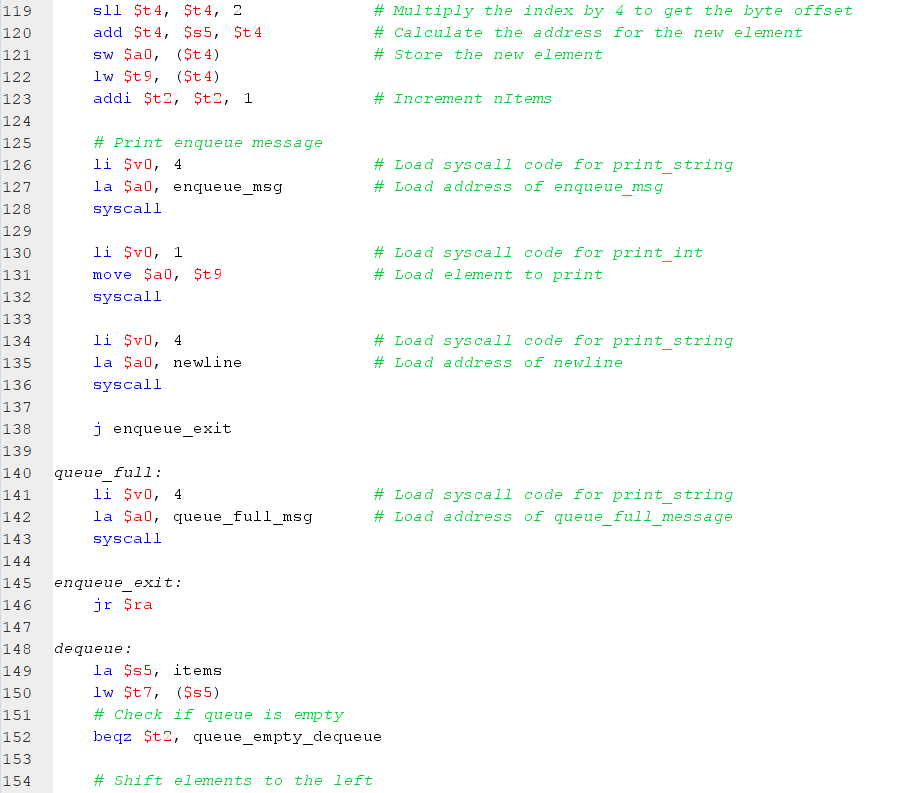
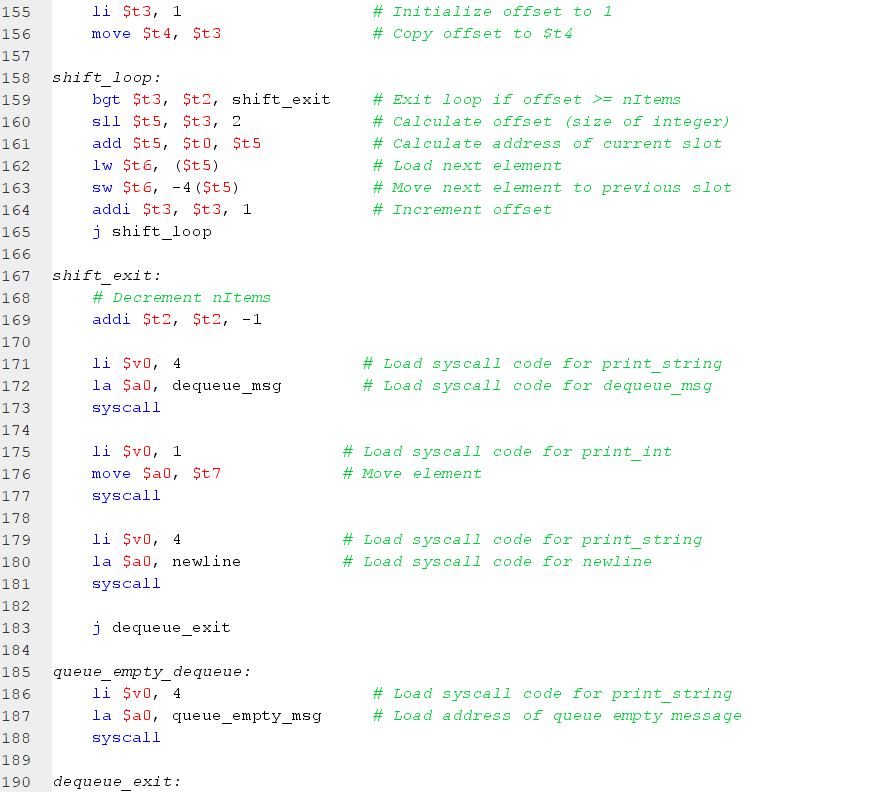
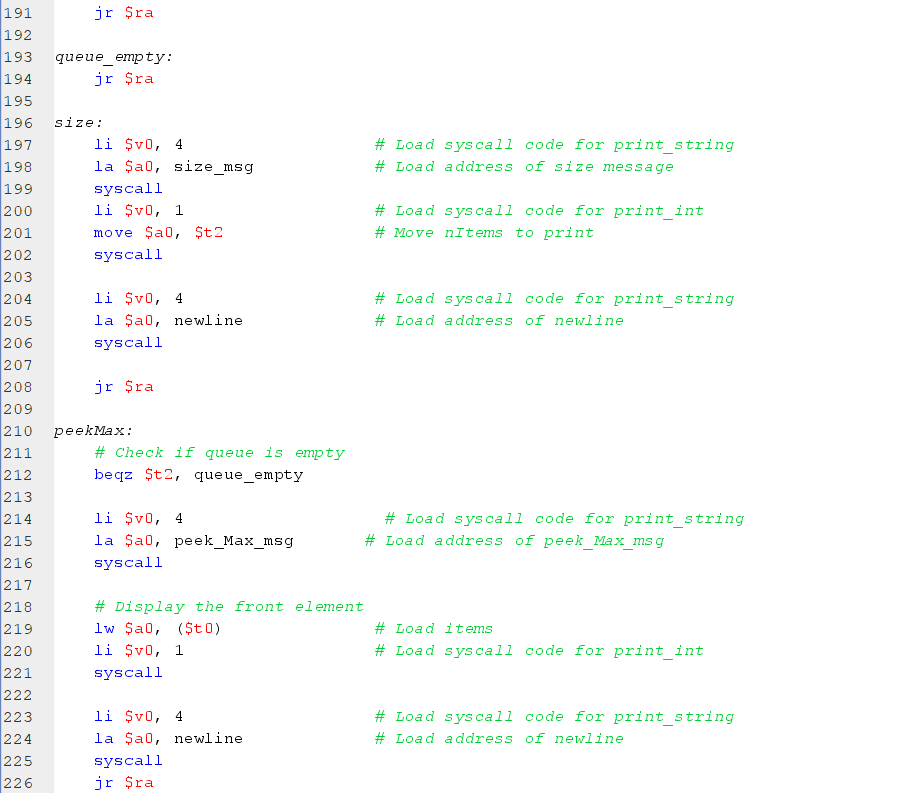
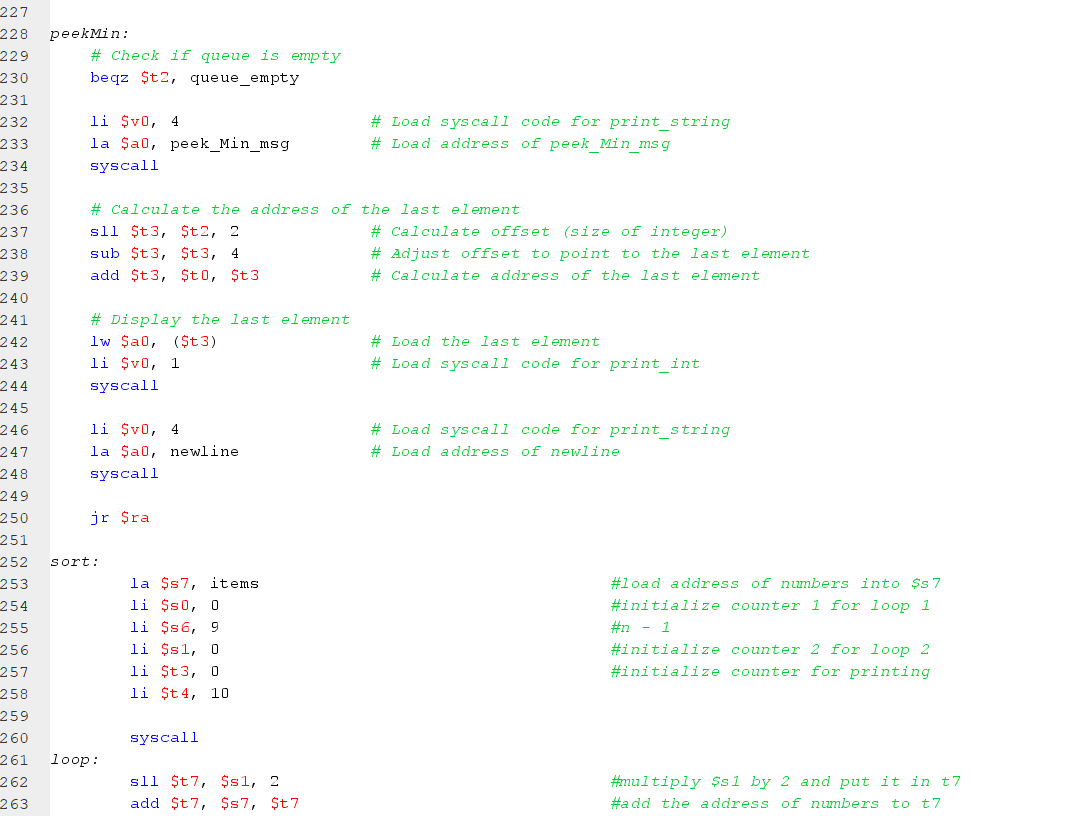
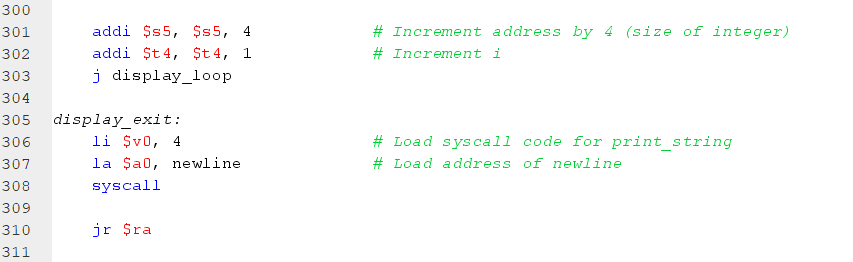
## Code:

### Java code





### Assembly code

# Life application: Bank Teller Services

## Application Statement

In banks, customers often wait in line for teller services like depositing, withdrawing, or transferring funds, or cashing checks. An array-based linear queue efficiently manages this queue by adding customers to the end and serving them in order. To enhance efficiency and scalability, banks can implement dynamic array resizing for flexible queue size, prioritize urgent transactions with a priority queue system, organize multiple queues for different transaction types, apply queue management algorithms to optimize service order, and employ parallel processing with multiple tellers for concurrent transactions. These strategies collectively streamline operations, reduce wait times, and improve customer satisfaction in banking environments.

## Implementation

### Working algorithm

1. The `.data` section defines the data variables and messages used in the program. It includes the `items` array to store the elements of the queue, `capacity` to store the maximum capacity of the queue, and `nItems` to store the number of elements currently in the queue. There are also several messages for displaying information about the queue.
2. The `.text` section contains the main program code, starting with the `main` label.
3. In the `main` function, the variables are initialized. The base address of the `items` array is loaded into `$t0`, the capacity is loaded into `$t1`, and the number of items is initialized to 0 by loading 0 into `$t2`.
4. The `enqueue` subroutine is called three times to add elements to the queue. The elements 101, 102, and 103 are enqueued by passing them in the argument `$a0` and calling the `enqueue` subroutine.
5. After enqueuing the elements, the `display` subroutine is called to print the current state of the queue.
6. The `dequeue` subroutine is called to remove an element from the front of the queue.
7. After dequeuing an element, the `display` subroutine is called again to print the updated state of the queue.
8. Finally, the program exits.

Here's a brief overview of the code's functionality:

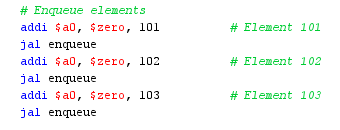
1. **enqueue**
   * + - The `enqueue` subroutine starts by checking if the queue is full. It compares the number of items (`$t2`) with the capacity (`$t1`) using the `bgt` (branch if greater than) instruction. If the queue is full, the program jumps to the `queue\_full` label and displays a message.
       - If the queue is not full, it proceeds to store the element in the next available slot in the `items` array. First, it calculates the offset by shifting the number of items left by 2 bits, which is equivalent to multiplying it by 4 (the size of an integer). The base address (`$t0`) is added to the offset to get the address of the next slot. The element is stored in the slot using the `sw` (store word) instruction. Then, the number of items is incremented by 1.
       - After enqueueing the element, it displays a success message by using the `li` (load immediate) instruction to set `$v0` to 4 (system call code for printing a string). The address of the `enqueue\_msg` string is loaded into `$a0`, and a system call is made to print the message.
       - The subroutine exits by jumping to the `enqueue\_exit` label.
2. **dequeue**

* The `dequeue` subroutine starts by loading the base address of the `items` array into register `$s5` and loading the value at the front of the queue into register `$t7`.
* It checks if the queue is empty by comparing the number of items (`$t2`) with 0 using the `beqz` (branch if equal to zero) instruction. If the queue is empty, the program jumps to the `queue\_empty\_dequeue` label and displays a message.
* If the queue is not empty, it shifts the elements in the array to the left to remove the first element. It uses a loop to iterate from offset 1 to the number of items. In each iteration, it calculates the address of the current slot, loads the next element, and stores it in the previous slot. This effectively shifts all the elements one position to the left.
* After shifting the elements, it decrements the number of items by 1.
* It displays a message indicating that a customer has been served and removed from the queue using the `li` instruction to set `$v0` to 4 (system call code for printing a string). The address of the `Customer\_served\_msg` string is loaded into `$a0`, and a system call is made to print the message.
* It displays the value of the served customer by using the `li` instruction to set `$v0` to 1 (system call code for printing an integer). The value of the served customer (`$t7`) is moved to `$a0`, and a system call is made to print the integer.
* Finally, it displays a newline character by using a similar process as step 5.
* The subroutine exits by jumping to the `dequeue\_exit` label.

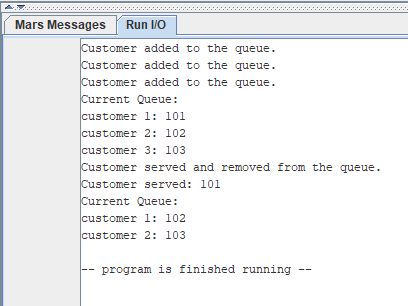
1. **display**

* subroutine is responsible for printing the current state of the queue. It checks if the queue is empty and, if not, iterates over the elements in the `items` array and displays them along with their respective indices.

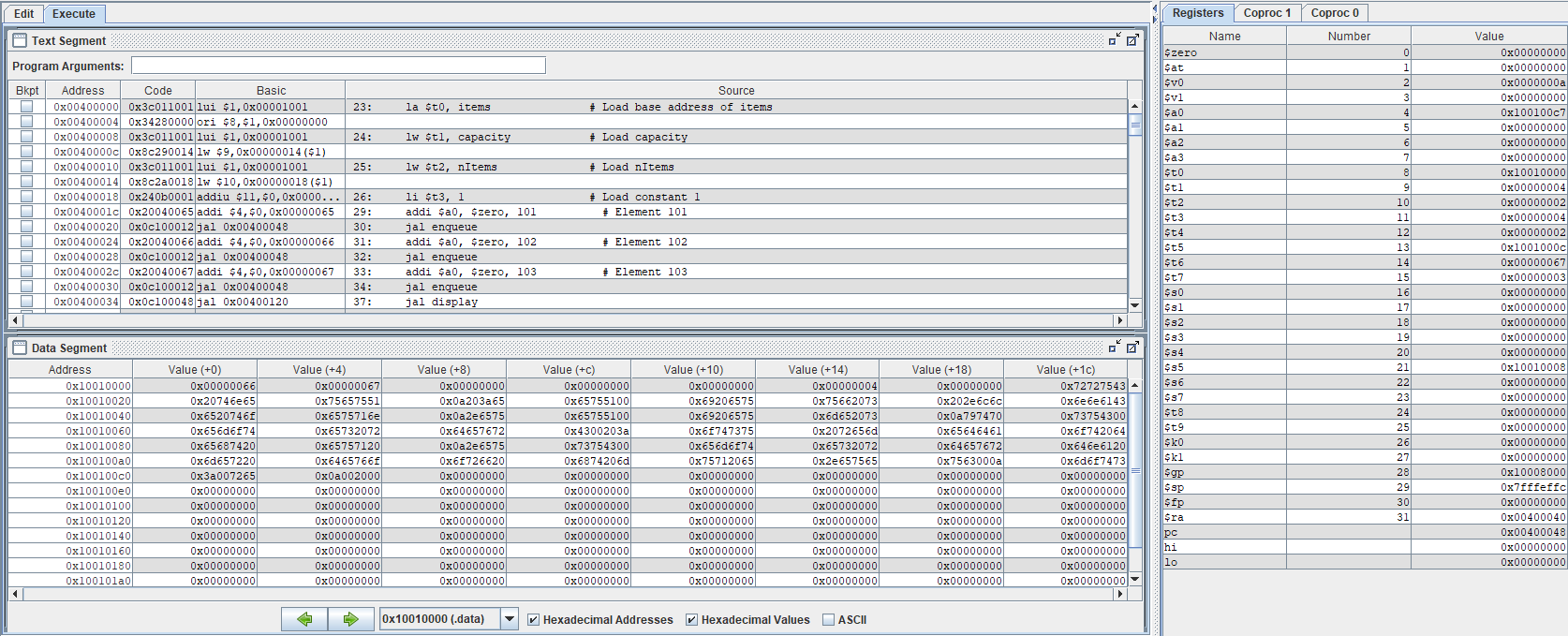
### Input



### Output

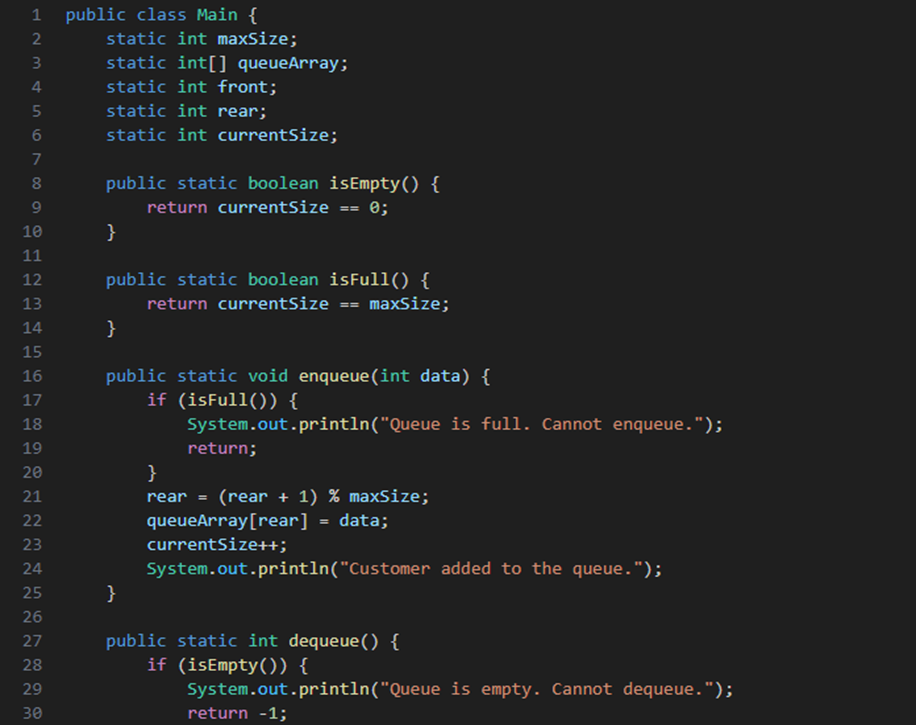


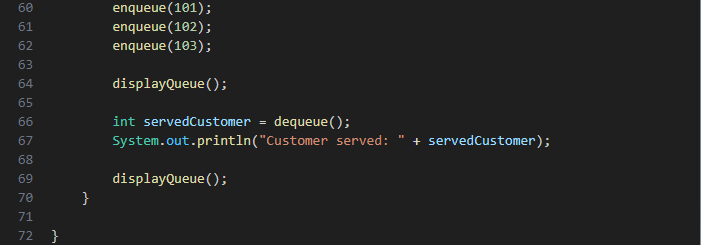
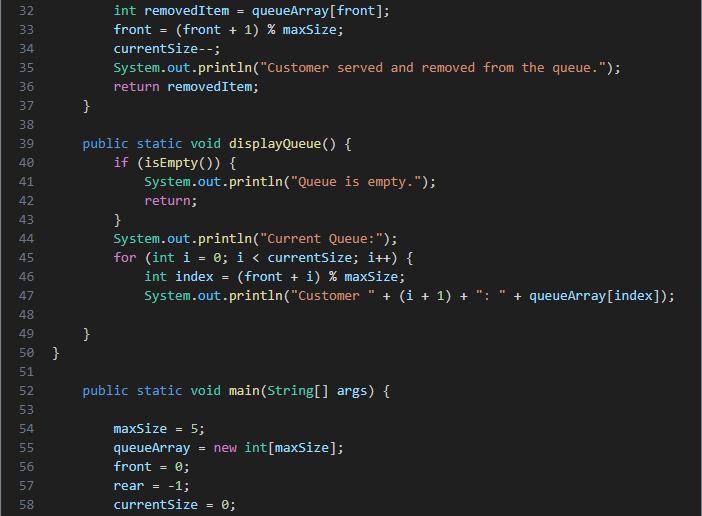
### Register values & values in memory segments



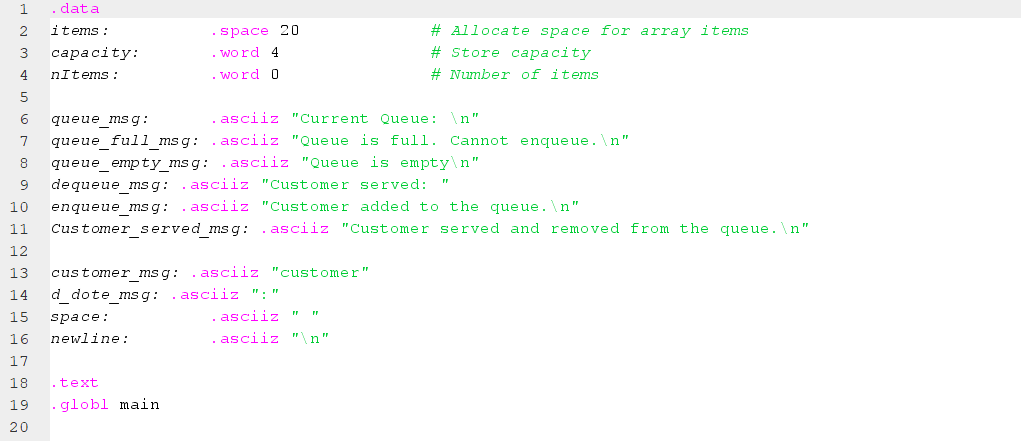
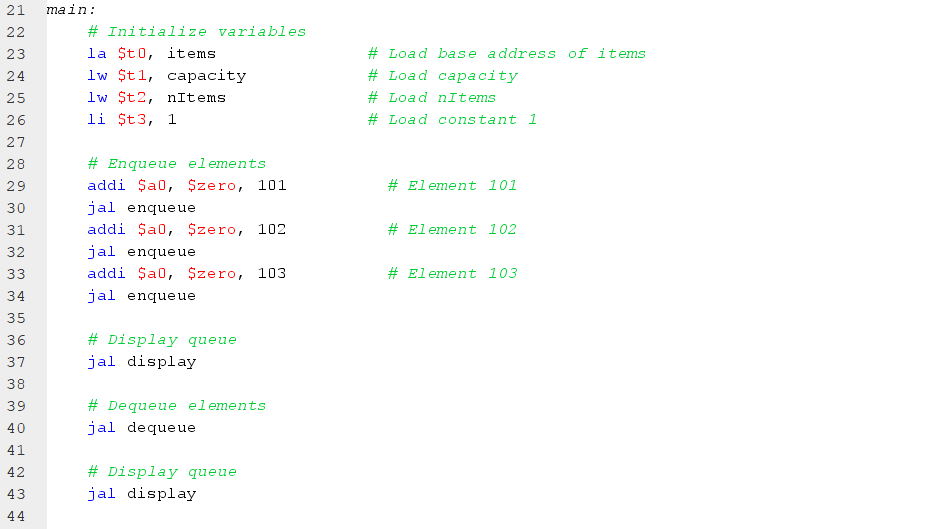
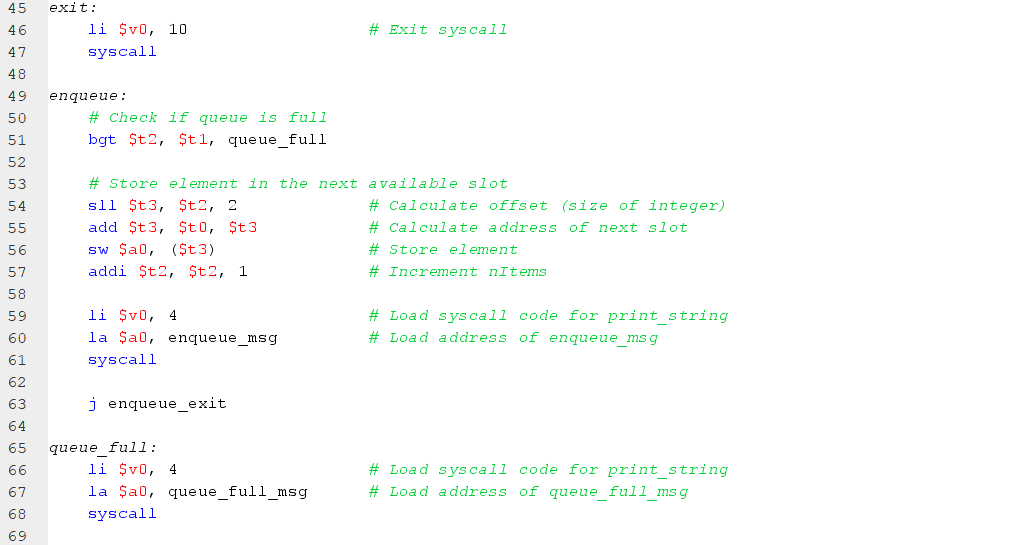
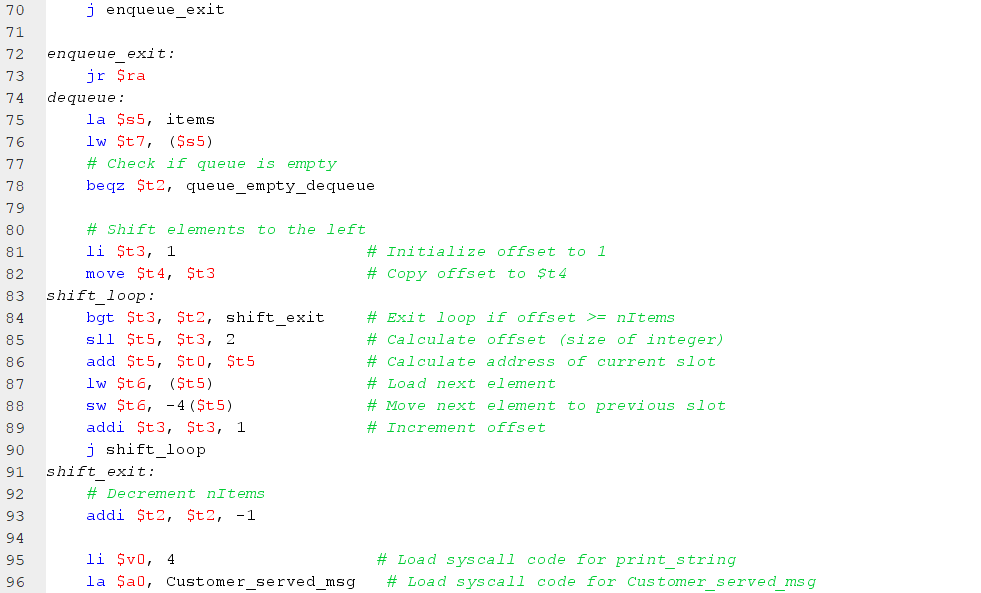
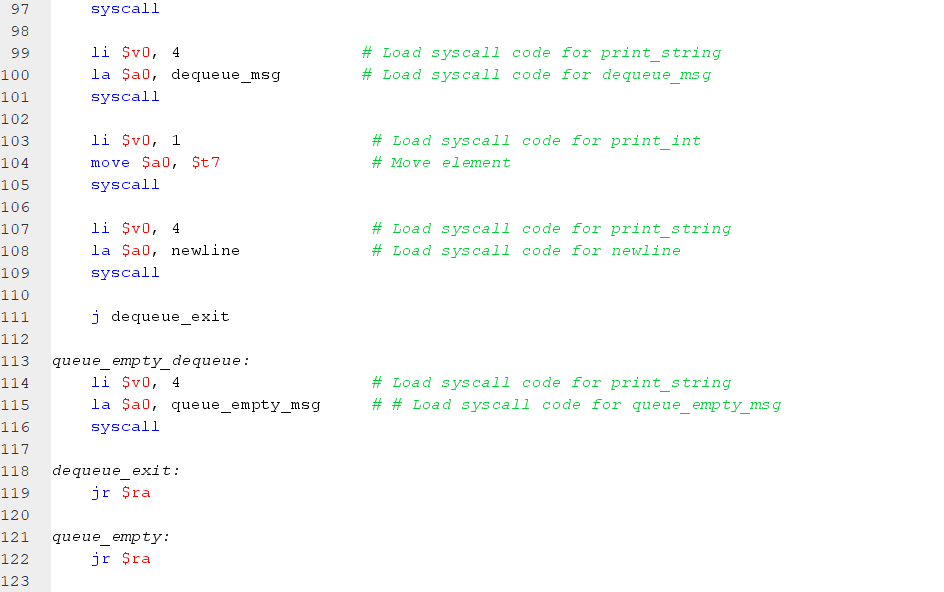
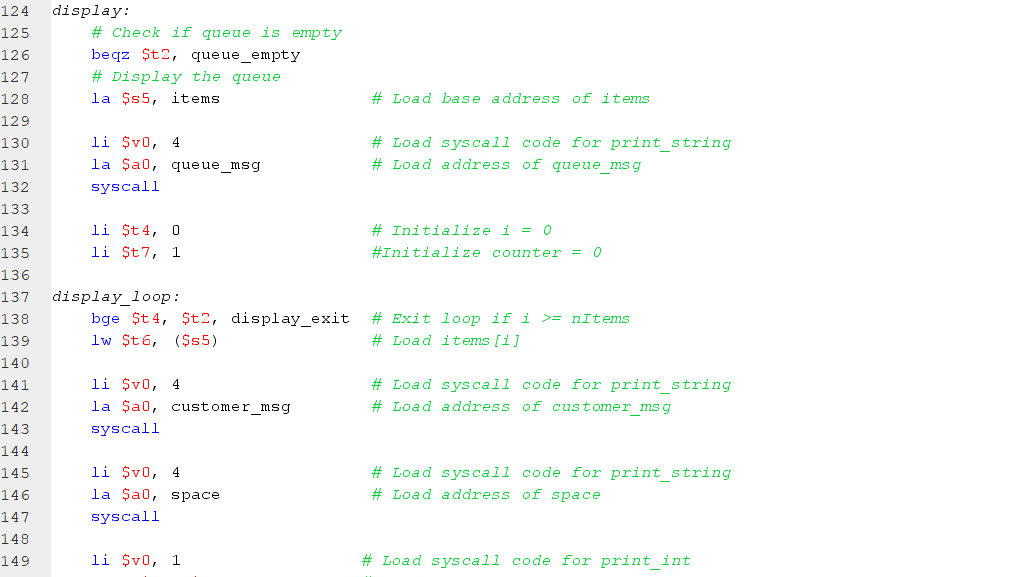
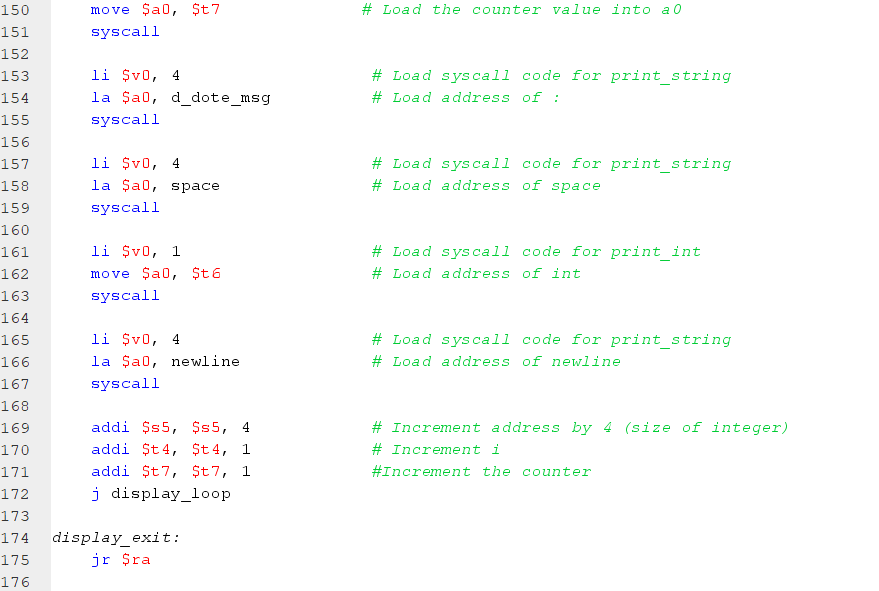
## Code

### Java code





### Assembly c ode

# Task Management

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name\task** | **Searching** | **Coding** | **Report** | **Presentation** |
| **Akram Khaled** | 16.67% | 16.67% | 30% | 3.33% |
| **George Emil** | 16.67% | 16.67% | 3.33% | 30% |
| **Abdelwahab Khaled** | 16.67% | 16.67% | 3.33% | 30% |
| **Ziad Omran** | 16.67% | 16.67% | 30% | 3.33% |
| **Ali El-Shaarawy** | 16.67% | 16.67% | 30% | 3.33% |
| **Kenzy Walid** | 16.67% | 16.67% | 3.33% | 30% |