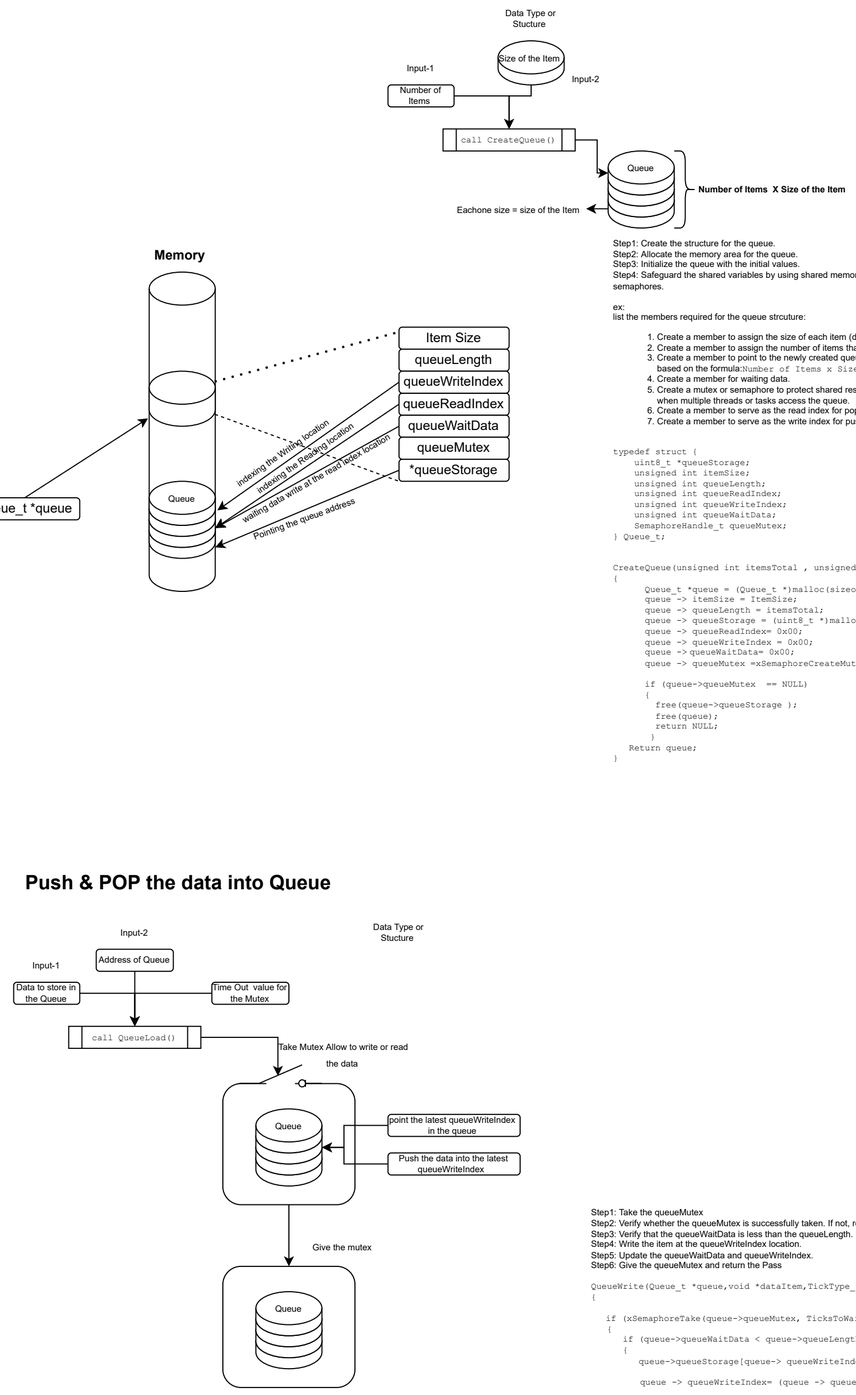
Create the Queue



Step1: Create the structure for the queue.
Step2: Allocate the memory area for the queue.
Step3: Initialize the queue with the initial values.
Step4: Safeguard the shared variables by using shared memory interfaces like mutexes or semaphores.

ex:
list the members required for the queue strcuture:

1. Create a member to assign the size of each item (data type or structure).
2. Create a member to assign the number of items that the queue will hold.
3. Create a member to point to the newly created queuestorage, with memory allocated based on the formula:`Number of Items x Size of Each Item.`
4. Create a member for waiting data.
5. Create a mutex or semaphore to protect shared resources and ensure thread safety when multiple threads or tasks access the queue.
6. Create a member to serve as the read index for popping data from the queue.
7. Create a member to serve as the write index for pushing data into the queue.

```
typedef struct {
    uint8_t *queueStorage;
    unsigned int itemSize;
    unsigned int queueLength;
    unsigned int queueReadIndex;
    unsigned int queueWriteIndex;
    unsigned int queueWaitData;
    SemaphoreHandle_t queueMutex;
} Queue_t;


CreateQueue(unsigned int itemsTotal , unsigned int ItemSize)
{
      Queue_t *queue = (Queue_t *)malloc(sizeof(Queue_t));
      queue -> itemSize = ItemSize;
      queue -> queueLength = itemsTotal;
      queue -> queueStorage = (uint8_t *)malloc(itemsTotal * ItemSize);
      queue -> queueReadIndex= 0x00;
      queue -> queueWriteIndex = 0x00;
      queue -> queueWaitData= 0x00;
      queue -> queueMutex =xSemaphoreCreateMutex();

      if (queue->queueMutex  == NULL)
      {
        free(queue->queueStorage );
        free(queue);
        return NULL;
       }
   Return queue;
}
```

Push & POP the data into Queue

Step1: Take the queueMutex
Step2: Verify whether the queueMutex is successfully taken. If not, return the mutex error.
Step3: Verify that the queueWaitData is less than the queueLength. If not, return the queue length error.
Step4: Write the item at the queueWriteIndex location.
Step5: Update the queueWaitData and queueWriteIndex.
Step6: Give the queueMutex and return the Pass

```
QueueWrite(Queue_t *queue,void *dataItem,TickType_t TicksToWait)
{

   if (xSemaphoreTake(queue->queueMutex, TicksToWait) == pdTRUE)
   {
      if (queue->queueWaitData < queue->queueLength)
      {
         queue->queueStorage[queue-> queueWriteIndex * queue-> ItemSize] = dataItem;

         queue -> queueWriteIndex= (queue -> queueWriteIndex+ 1) % queue -> queueLength;

         queue -> queueWaitData++;

         xSemaphoreGive(queue->queueMutex);

         return Pass;
       }

      xSemaphoreGive(queue->queueMutex);

      return QueueError;

    }

   Return MutexError;
}
```

Step1: Take the queueMutex
Step2: Verify whether the queueMutex is successfully taken. If not, return the mutex error.
Step3: Verify that the queueWaitData is less than the queueLength. If not, return the queue length error.
Step4: Write the item at the queueReadIndex location.
Step5: Update the queueWaitData and queueReadIndex.
Step6: Give the queueMutex and return the Pass

```
QueueWrite(Queue_t *queue,void *dataItem,TickType_t TicksToWait)
{

   if (xSemaphoreTake(queue->queueMutex, TicksToWait) == pdTRUE)
   {
      if (queue->queueWaitData > 0)
      {
         dataItem = queue->queueStorage[queue-> queueReadIndex * queue-> ItemSize];

         queue -> queueReadIndex = (queue -> queueReadIndex + 1) % queue-> queueLength;

         queue -> queueWaitData--;

         xSemaphoreGive(queue->queueMutex);

         return Pass;
       }

      xSemaphoreGive(queue->queueMutex);

      return QueueError;

    }

   Return MutexError;
}
```