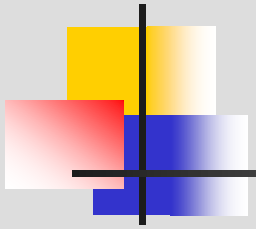
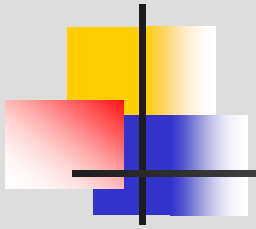


Data structures

- Assembly
  - Basit veri tipleri (characters, integers, floating point)
- High level Programming
  - Lists, trees, stacks, files, databases…
- High level structures low level structure lardan elde edilir (abstraction)



MEMORY (BELLEK)

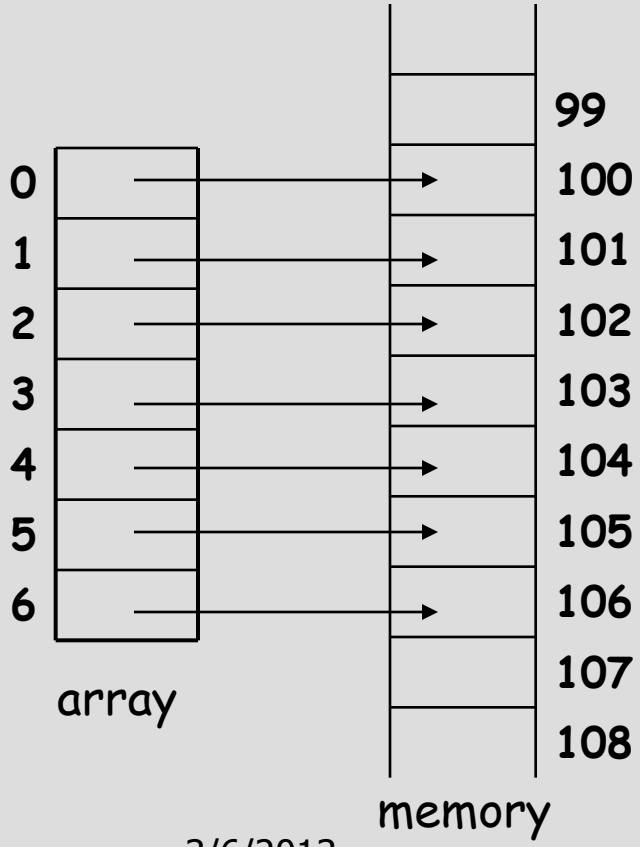
- Memory cells (bellek gozlerinden) olusur.
- Bellek cells lerden olusan tek boyutlu bir dizi olarak dusunulebilir.
- Her bir cell e bir adres verilir.
  - Adresler positive integer larla temsil edilirler.

Memory Addresleme

- Addresleme
  - Byte Addressing
    - Eger adreslenen cells lerin boyu 1 byte (8 bit) ise, buna byte addressing denir.
  - Word Addressing
- Word
  - Integer lari temsil etmek icin kullanilan bellek alanina denir.
  - Cogu bilgisayarlarda 32 bit lik alan word olarak kabul edilir.
- Endian
  - Big-endian
  - Little-endian
- Alignment
  - simplicity

Arrays (Characters)

ar: array [0..6] of char ⟹ ar: .byte 0:7

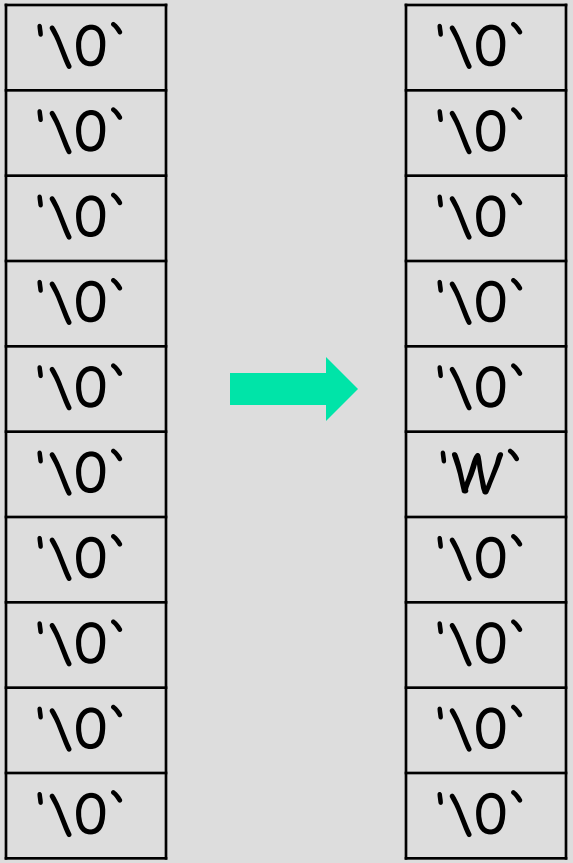


ar: array [first_index .. last_index] of char

ar: .byte 0: (last_index-first_index +1)

ar[i] → m[ar + i – first_index]

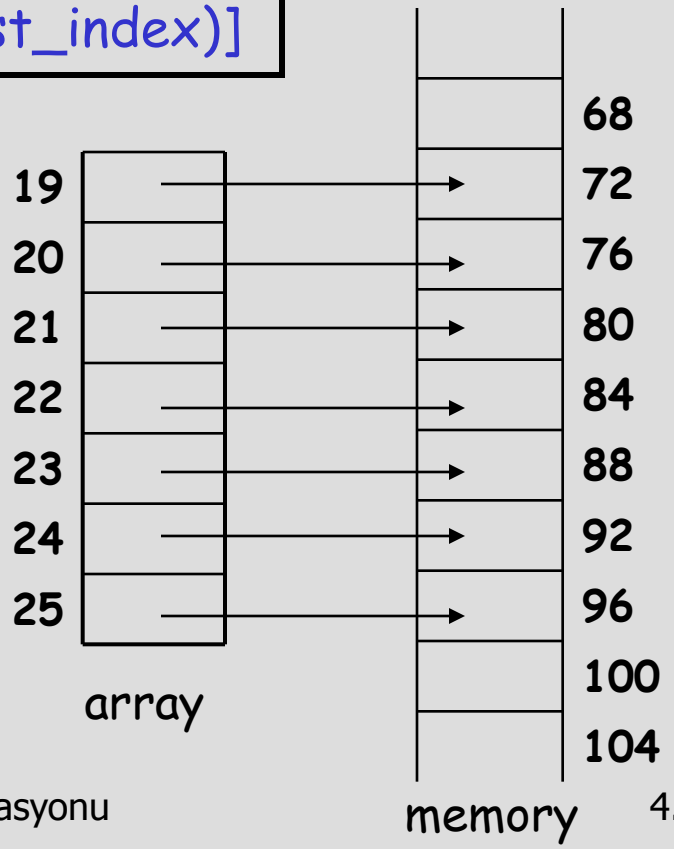
```
        .data
        ar:     .byte  0:10

        .text

la      i, ar
add     i, i, 5
move    m[i], 'W'
```

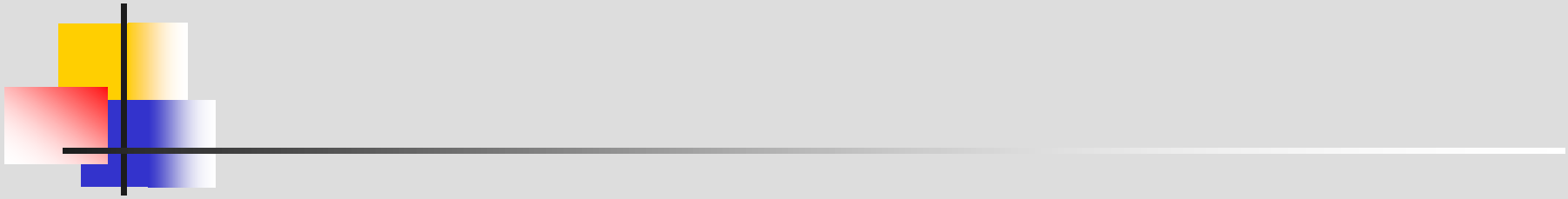


Array (Integers)

```
ar: array [first_index .. last_index] of int
ar:       .byte    0: (last_index-first_index +1)
ar[i] → m[ar + size_of_element . (i – first_index)]
```

```
ar: array [19.. 25] of int
ar:       .byte    0: (7)
ar[23] → m[88]
```





```
ar:     .byte  0:40
ar:     .word  0:10
ar:     .space  40
```

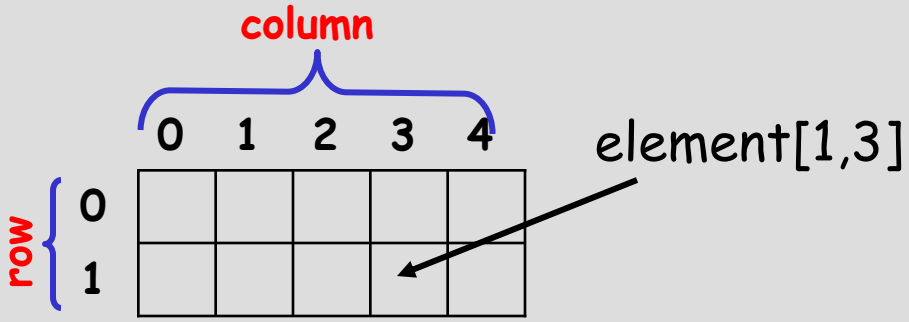
Two-Dimensional Arrays

```
ar_2D:  array [first_row .. last_row,
               first_col .. last_col ] of char;
```

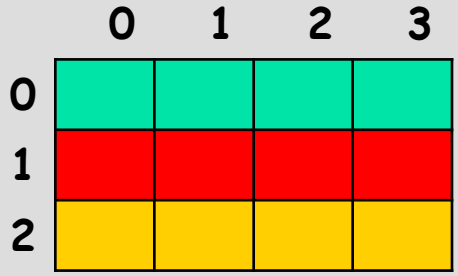
number_of_elements = (*last_row – first_row + 1*) . (*last_col – first_col + 1*)
array_size = (*number_of_elements*) . (*size_of_element*)

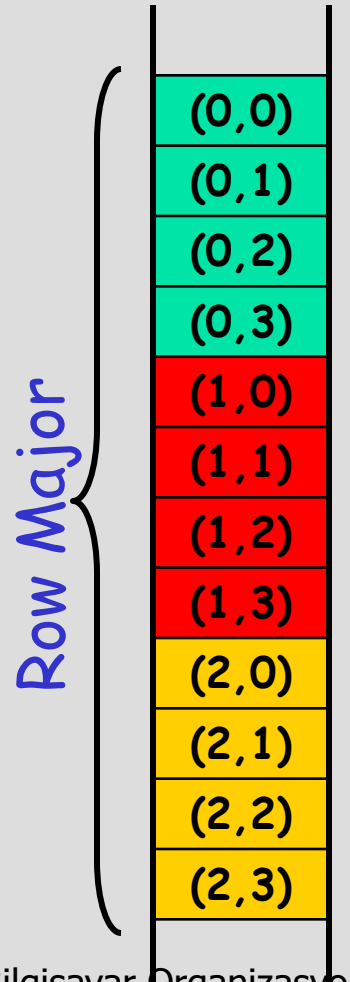
```
ar_2D:  .space  array_size
ar_2D:  <type>  initial_value : number_of_elements
```

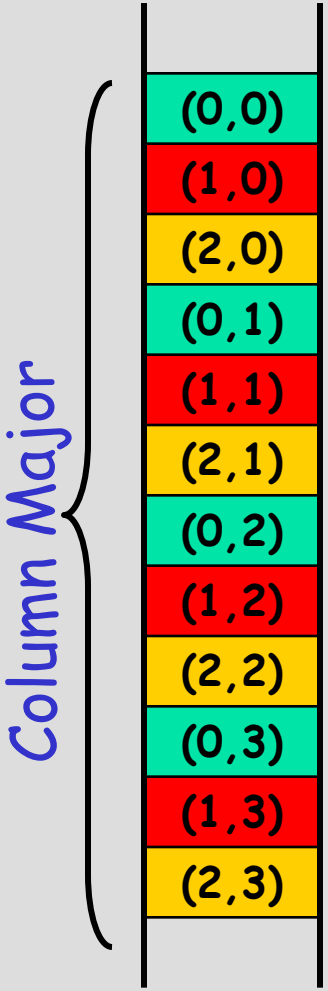
burada <type> .byte veya .word olarak belirlenir.



Storage Order

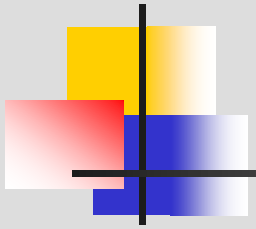






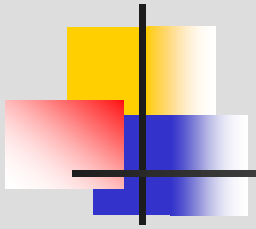
```
.data
        array:                  .space   400     # bytes for a 10x10
                                                 # array of integers
        row:                    .word
        col:                    .word
        base:                   .word
        address:                .word
        elements_in_row:        .word    10
        elements_in_column:     .word    10
        size:     .word    4
        .text
        move      row, 0
        move      col,  2
        la        base, array
loop:   beq       row, elements_in_col, next
        mul       address, row, elements_in_row
        add       address, address, col
        mul       address, address, size          # 4 bytes in each element
        add       address, address, base          # address of the desired element
        move      M[address], 0                   # clear element
        add       row, row, 1                     # set up for the next row
        b         loop
next:
```

Dizinin ikinci kolonunun sifirlanmasi



STACKS

- Stack icin LIFO (Last-In-First-Out) deyimi de kullanilir.
- Genelde verilerin kullanilis sirasi uretim sirasinin tersi durumlarinda kullanilir.
- Stack Opereations
  - Push : Stack in ustune data yi yerlestirme
  - Pop: Stack ustundeki data yi cekme
  - Empty: Stack in bos olup olmadigi kontrolu
  - Full: Stack in dolu olup olmadigi kontrolu

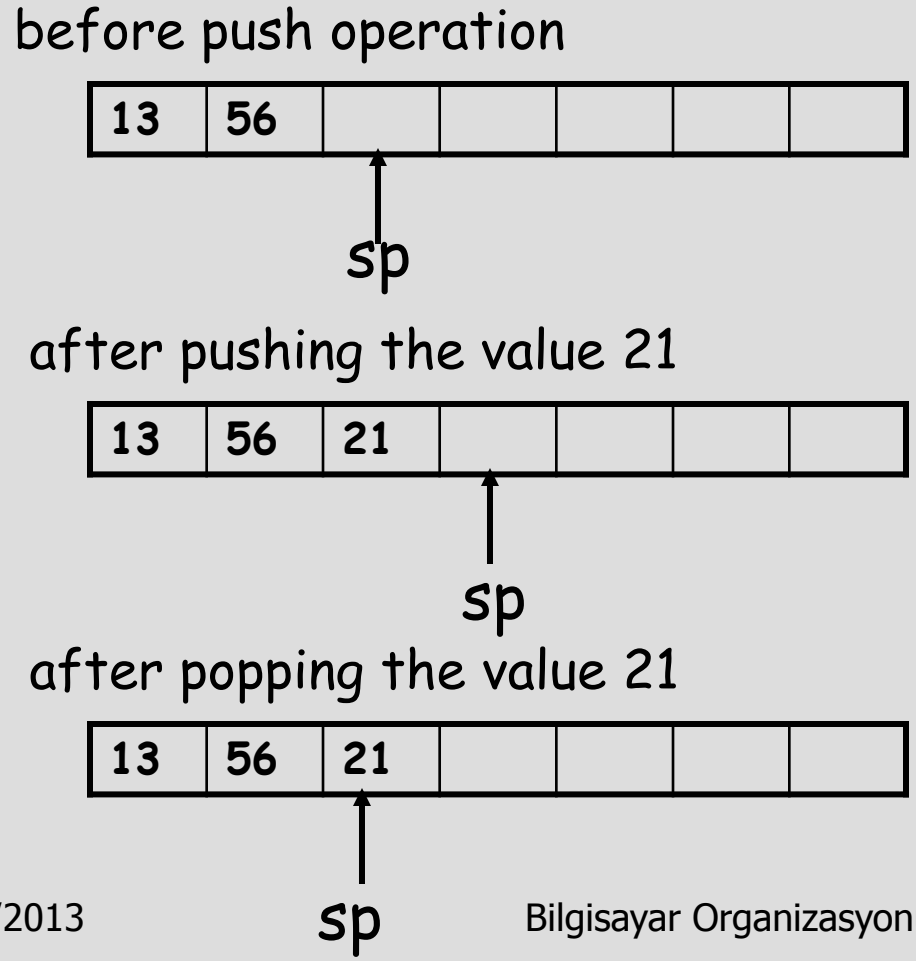


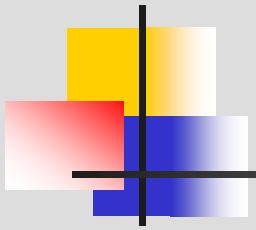
```
stack: .word  0:maxstacksize
sp:           .word  stack
```

```
push:
move M[sp], x
add    sp, sp, 4
```

```
        .data
stack: .word  0:maxstacksize
sp:           .word

        .text
        la sp, stack
```

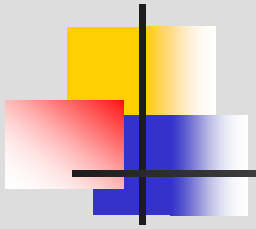
```
pop:
add    sp, sp, -4
move  x, M[sp]
```


before push operation
13
56
sp
after pushing the value 21
13
56
21
sp
after popping the value 21
13
56
21
sp



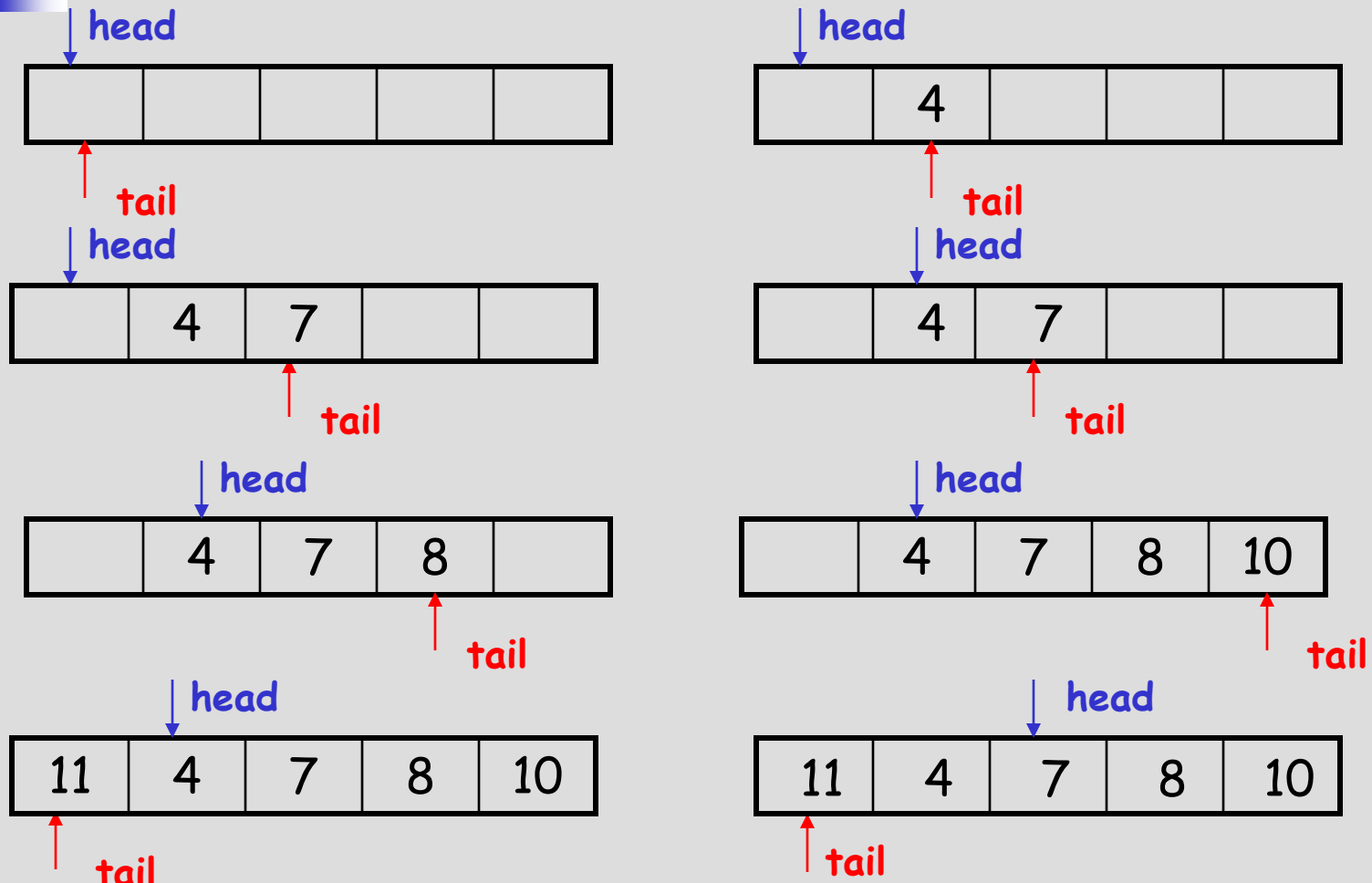
```
            .data
stack:      .byte   0:50
sp:         .word   stack     # top of stack pointer
bottom:     .word             # bottom of stack
bias        .word   48        # decimal value of ASCII character '0'
number      .word
digit       .word

            .text
loop_top:   rem     digit, number, 10
            add     digit, digit, bias
            move    m[sp], digit        # push character onto stack
            add     sp, sp, 1
            div     number, number, 10
            bgtz    number, loop_top
            la      bottom, stack
print_it:   add     sp, sp, -1          # pop character off stack
            putc    m[sp]
            bgt     sp, bottom, print_it
```

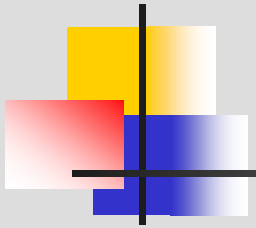
Integer bir sayinin ekrana yazilmasi



QUEUES

- Verilerin kullanim sirasi uretildikleri sirayla ayni ise kuyruk veri yapisi kullanilir.
- First-In-First-Out (FIFO)
- Operations
  - enqueue (veriyi kuyruga yerlestirmek)
  - dequeue (veriyi kuyruktan cikarmak)
- Circular buffer kullanilirsa, kuyruga en fazla kapasitesinin bir eksigi sayida veri konabilir.


head
tail
head
4
7
tail
head
4
7
8
tail
head
11
4
7
8
10
tail
head
4
tail
head
4
7
tail
head
4
7
8
10
tail
head
11
4
7
8
10
tail



```
                .data
queue:          .byte   0:64    # Array to hold queue
queueaddr:      .word           # Address of array holding queue
head:           .word   0       # head of offset
tail:           .word   0       # tail of offset
linenumber:     .byte   0       # phone line to be enqueued
nextline:       .byte           # phone line to be dequeued
addr:           .word
newline:        .byte
string1:        .asciiz "Which line is ringing? "
string2:        .asciiz "The next line to be answered is "
string3:        .asciiz "Enqueuing line "
empty:          .asciiz "No calls waiting. "
full:           .asciiz "ERROR: Queue is full. Exiting program. "
```

```
                .text
__start:        la      queueaddr, queue
loop:           puts    string1
                get     linenumber
                beq     linenumber, '\n', dequeue
                get     newline
enqueue:        puts    string3
                put     linenumber
                put     '\n'
                add     tail, tail, 1
                rem     tail, tail, 64
                beq     tail, head, full_queue      # branch if tail+1 == head
                add     addr, queueaddr, tail
                move    m[addr], linenumber
                b       loop
dequeue:        beq     head, tail, empty_queue     # branch if head == tail
                add     head, head, 1
                rem     head, head, 64
                add     addr, queueaddr, head
                move    nextline, m[addr]
                puts    string2
                put     nextline
                put     '\n'
                b       loop
empty_queue:    puts    empty
                put     '\n'
                b       loop
full_queue:     puts    full
                done
```