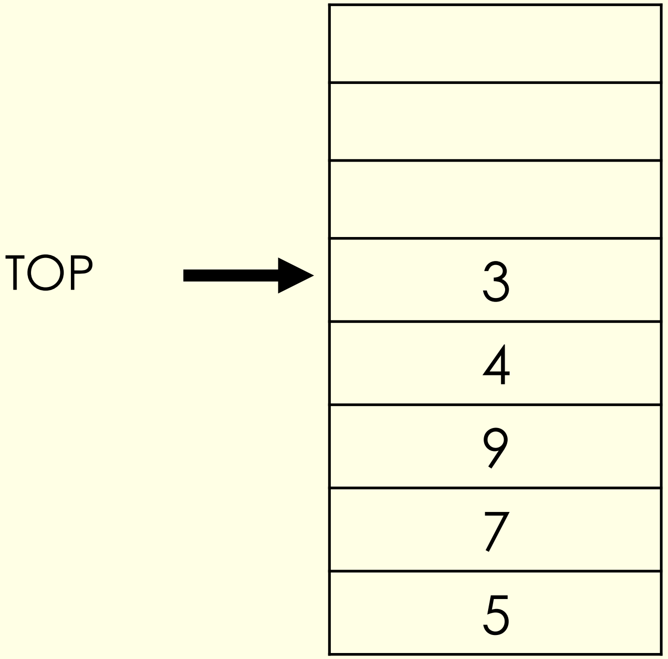
Stacks and Queues

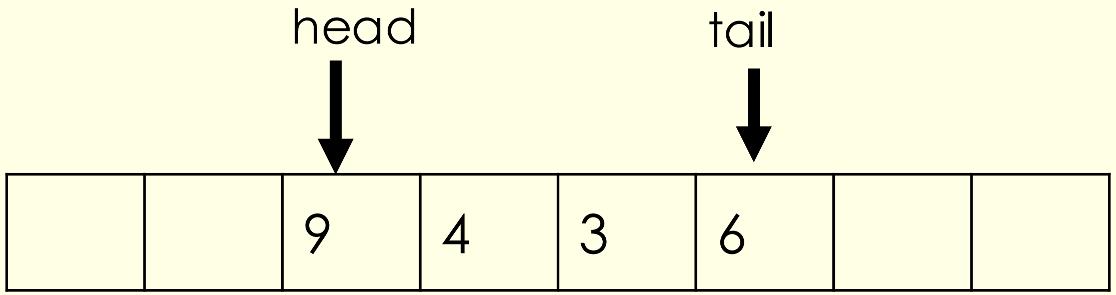
• Be familiar with the concept and uses of a stack • Be able to describe the creation and maintenance of data within a stack • Be able to describe and apply the following operations: push, pop, peek (or top), test for empty stack, test for full stack • Be able to explain how a stack frame is used with subroutine calls to store return addresses, parameters and local variables	• Be familiar with the concept and uses of a queue • Describe the creation and maintenance of data within a queue (linear, circular, priority) • Describe and apply the following to a linear, circular and priority queue o add an item o remove an item o test for an empty queue o test for a full queue

Stacks and Queues

last in first out (LIFO)



First in first out (FIFO)



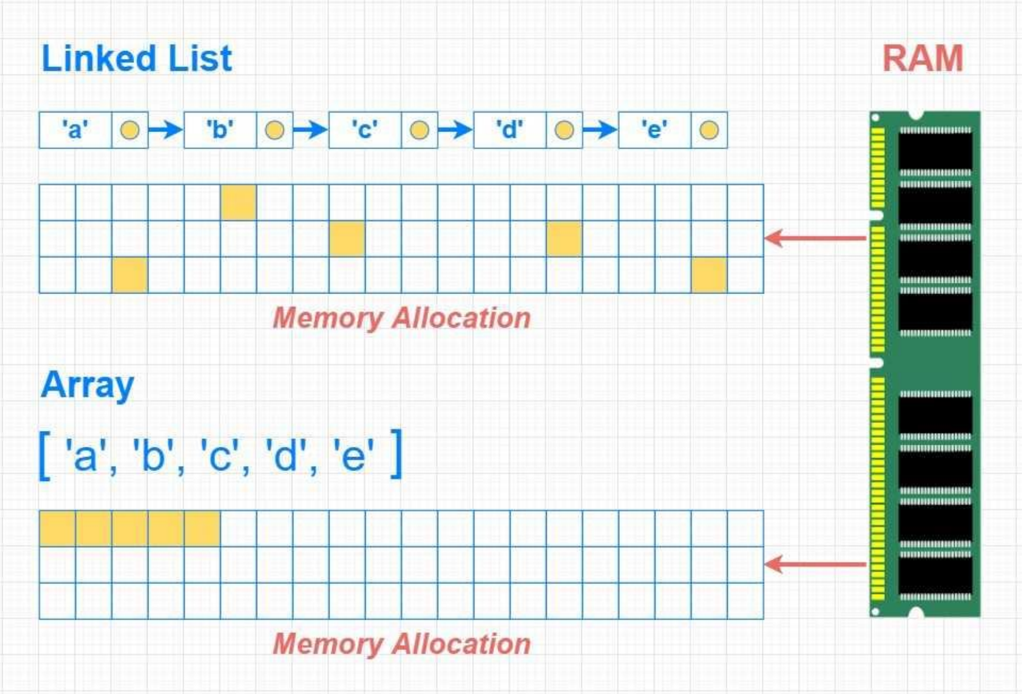
Stacks and Queues

- **A linear data structure** represented by a sequential collection of elements in a **fixed order**
- **Dynamic size.**
- Contain elements of **different data types.**
- Random access of elements are not allowed
- Implement using array or linked list

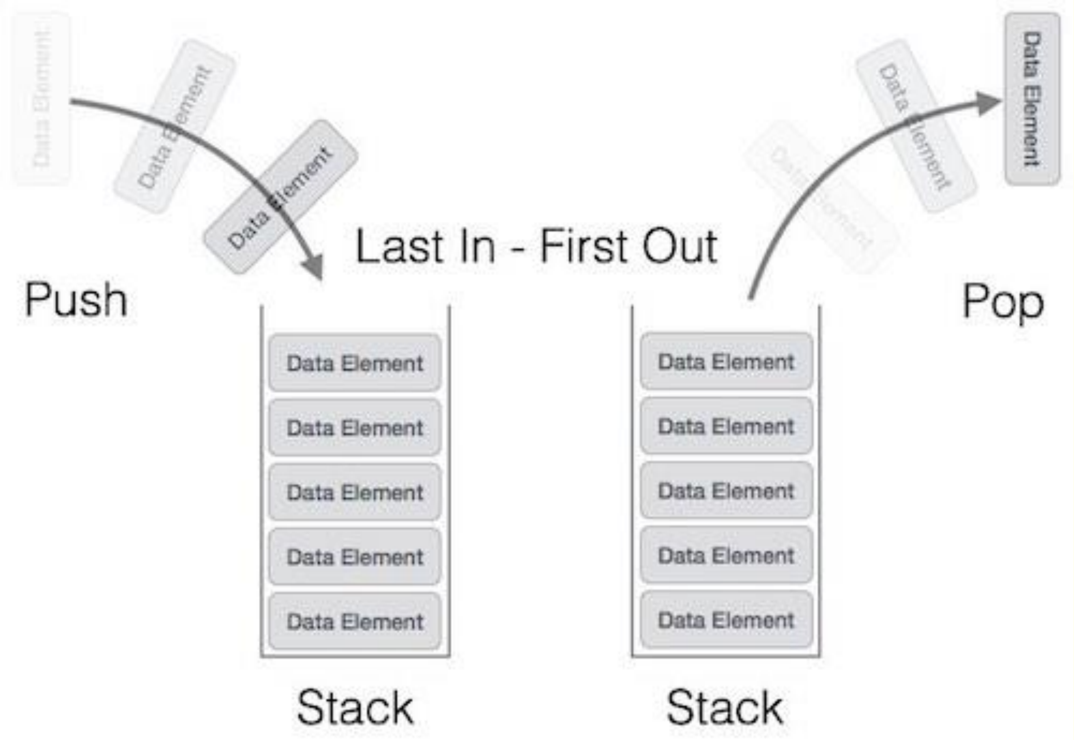
Stack	Queue
LIFO – Last In, First Out	**FIFO** – First In, First Out
Only the **top** element can be accessed	Only the **head** element can be accessed
It has only one pointer- the **stack pointer** This pointer indicates the address of the topmost element or the last inserted one of the stack.	Two pointers – **head** and **tail**
push() – Pushing (storing) an element on the stack. **pop()** – Removing (accessing) an element from the stack. **peek()** – get the top data element of the stack, without removing it. **isFull()** – check if stack is full. **isEmpty()** – check if stack is empty	enqueue() – Adds to the tail dequeue() – Removes from the head isFull() – check if queue is full isEmpty() – check if stack is empty
Browser history (Back button) Undo operations in text editors Function call stack (program execution)	Printer queue (print jobs in order) Customer service queue Task scheduling (e.g. CPU process queue)

Implementing Stacks and Queues

- Using an array (list in python)
  - **Pros:** Easy to implement.
    Cons: It is not dynamic. It doesn't grow and shrink depending on needs at runtime.
- Using linked list
  - **Pros:** The linked list implementation of a stack can grow and shrink according to the needs at runtime.
    Cons: Requires extra memory due to involvement of pointers.



Stack Push and Pop



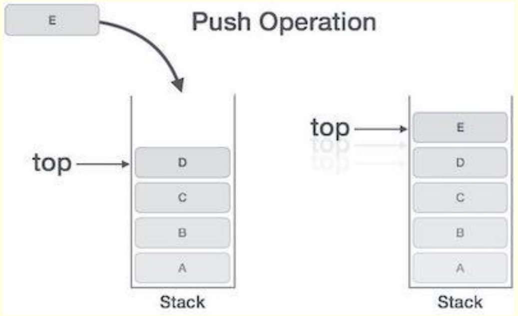
Stack PUSH Algorithm

Step 1 – Checks if the stack is full.

Step 2 – If the stack is full, produces an error and exit.

Step 3 – If the stack is not full, increments **top** to point next empty space.

Step 4 – Adds data element to the stack location, where top is pointing.

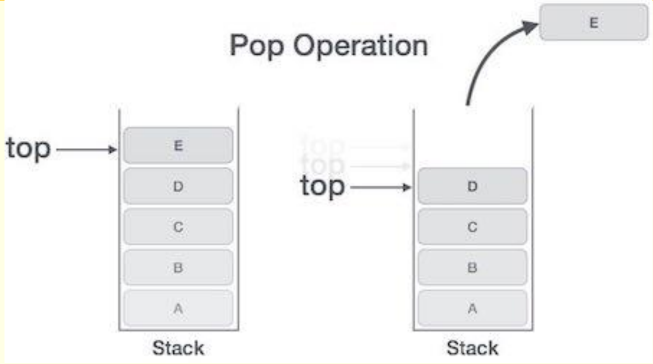


isFull() function

You could use:
if top >= MAX
if stack.length >= MAX

```
#If the stack is full an error message will be generated
if isFull() then
        print 'stack overflow'
else
        # else add 1 to the stack top pointer
        top=top+1
        #insert new item to the top of the stack
        stack[top] = item
```

Stack Pop Algorithm



Step 1 – Checks if the stack is empty.

Step 2 – If the stack is empty, produces an error and exit.

Step 3 – If the stack is not empty, accesses the data element at which **top** is pointing.

Step 4 – Decreases the value of top by 1.

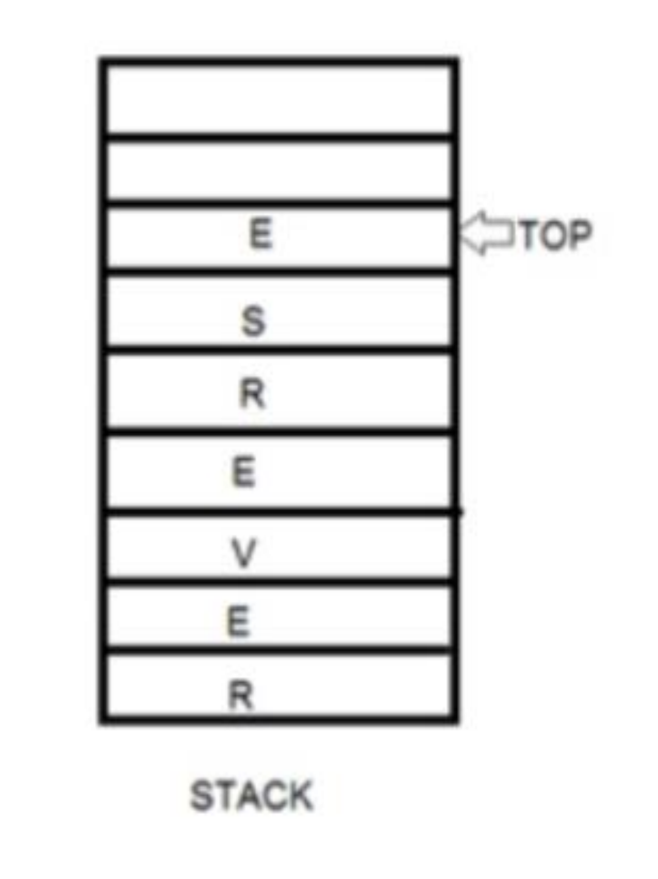
isEmpty() function

You could use
if top == -1:

```
#If the stack is empty an error message will be generated
if isEmpty() then
        print 'stack underflow'
else
        # else get the item to pop
        item = stack[top]
            #change the top of the stack
        top=top-1
        return item
end if
```

Display items in Stack algorithm

```
if isEmpty() then
        print 'stack empty'
else
        #only loop though the list from
              # the top to 0
        for i=top to 0
            print s[i]
        next i
end if
```



A function, push, can be used to add a character to a stack. For example:

theStack.push("H")

places the character H onto the stack, theStack.

A procedure, pushToStack, takes a string as a parameter and pushes each character of the message onto the stack, messageStack.

Complete the procedure below.

Add comments to explain how your code works.

procedure pushToStack(message)

1 mark per bullet to max 5

- Use of appropriate loop
- Correct end condition (length of message)
- Correct use of .push with messageStack
- Accessing substring (or equivalent) correctly
- Appropriate comment(s)

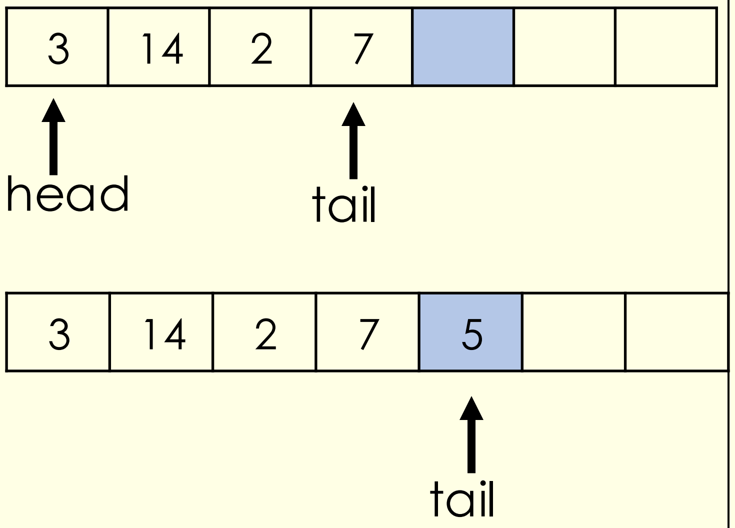
```
procedure pushToStack(message)
   for x = 0 to message.length() //loop through each
                                 //letter
      messageStack.push(message.substring(x,1)) //take
            //each character and push onto stack
   next x  //move to next letter
endprocedure
```

Here is an example of an exam question where you only need to use the procedures push() or pop()

Enqueue Algorithm in a linear queue

1. Check if queue is full
2. If full output error and stop
3. Else increment tail pointer
4. Insert new data item into the tail pointer position.

```
# error message if queue is full
if isFull() then
    print ("Overflow")
else
    tail = tail + 1
    queue[tail] = data
end if
```



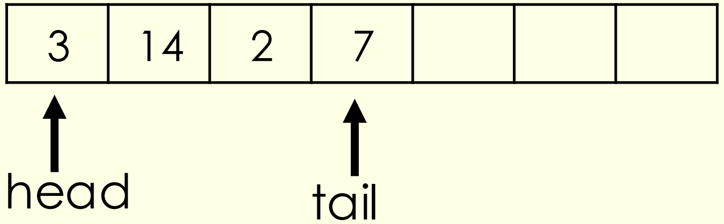
Working out the isFull() function can be done in a number of different ways, for example:
if tail == maxsize – 1
Or
by using a counter size to keep a track of
data items
if size == MAX

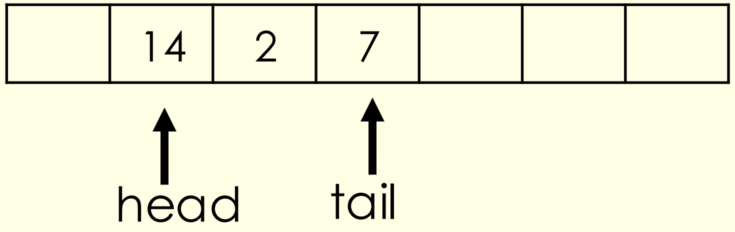
Dequeue Algorithm in a linear queue

1. Check if queue is empty
2. If empty output error and stop
3. Else copy data from the head pointer position
4. Increment head pointer
5. Return data

```
if isEmpty () then
    print ("Queue is empty")
else
    data = queue[head]
    head = head + 1
    return data
```

You will need to also reset the head to 0 and tail to -1





Working out the isEmpty() function

It can be done in a number of different ways, for example:

```
if head > tail
   head = 0
   tail = -1
```

Or by using the size counter :

```
if size = 0
```

Displaying a linear queue

```
#only loop though the list from the start to the end pointer
    for i=head to tail
        print q[i]
    next i
```

Lets now look at a circular queue …

- This means data is added at the end of the queue can be stored in locations vacated at the start of the queue.

	12	3	7	8	23	
	↑ head				↑ tail	

Enqueue 9

	12	3	7	8	23	9
	↑ head					↑ tail

Dequeue

		3	7	8	23	9
		↑ head				↑ tail

Enqueue 15

15		3	7	8	23	9
↑ tail		↑ head				

Notes for exam

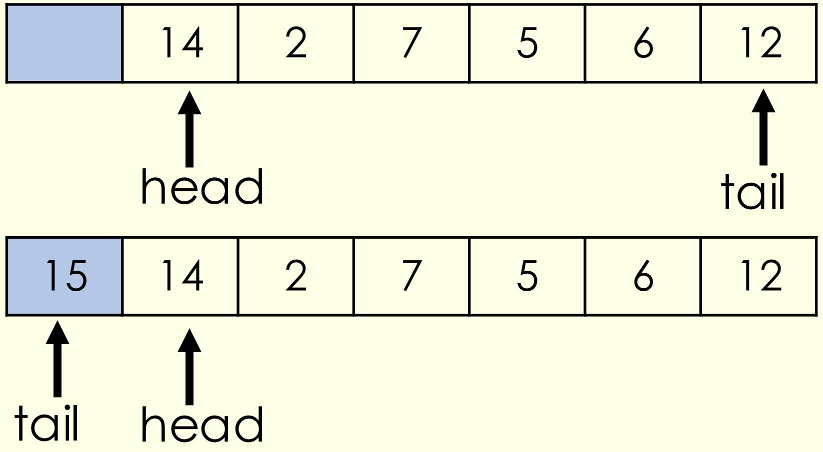
Start and end pointers can be either way round …

Pointers might be referred to as head/tail or front/rear

It can either be a circular and linear queue – it will specify this in the exam question

Algorithm for pushing an item to a circular queue (enqueue)

1. Check if queue is full
2. If full output error and stop
3. Else if tail is equal to maxsize set tail pointer to 0
4. Else increment tail pointer by 1
5. Insert new data item into the tail pointer position.
6. Increment size by 1




```
# error message if isFull is True
If isFull() then
    print overflow
else
       if tail = maxsize - 1
           tail= 0
       else
           tail = tail + 1
       queue[tail] = data
       size =size + 1
end if
```

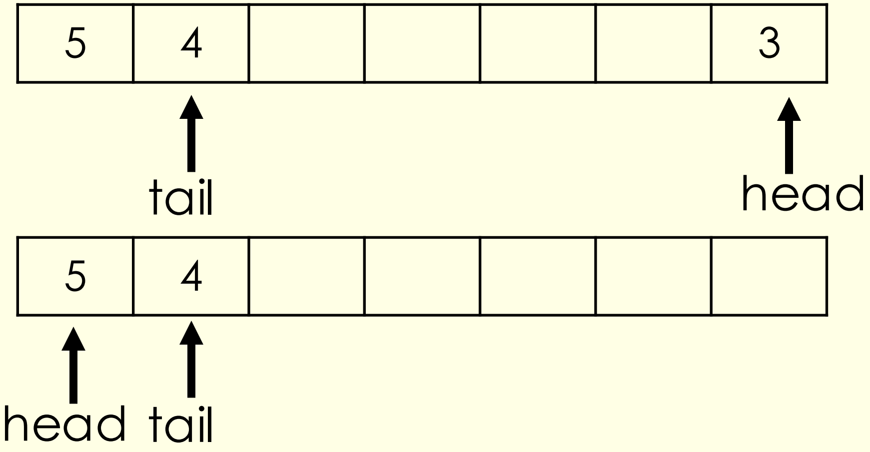
isFull() function may look something like this. The program uses a variable, i.e. size that keeps track of adding and removing items from the queue, so:

```
if size == maxsize:
    return True
```

Algorithm of popping from a circular queue (dequeue)

1. Check if queue is empty
2. If empty output error and stop (reset head and tail pointers)
3. Else copy data from the head pointer position
4. If head pointer is equal to maxsize then resent head pointer to 0
5. Else increment head pointer by 1
6. Decrement size by 1
7. return data

```
# error message if queue is empty
if isEmpty() then
   print empty
   head = 0
   tail = -1
else
   data = queue[head]
   if head = MAXSIZE -1 then
        head = 0
   else
        head = head +1
   end if
   size = size - 1
   return data
```

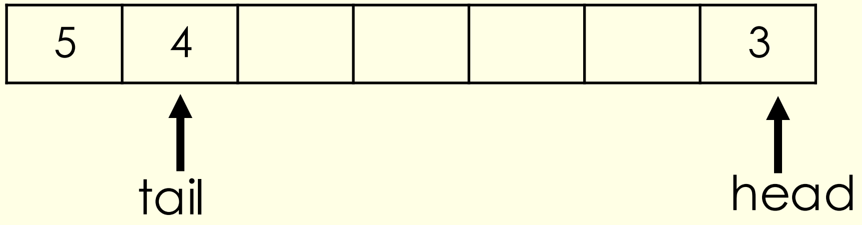


Working out the isEmpty() function can be done in a number of ways, for example, the program uses a variable, i.e. size that keeps track of adding and removing items from the queue, so:

```
if size = 0 then
     return True
```

Displaying a circular queue

```
If isEmpty() then
    print("Queue is empty")
else
    temp = head
    for i = 0 to size - 1
        print q[temp]
        temp = temp + 1
        if temp == maxsize then
            temp = 0  // wrap around manually
        end if
    end for
end if
```



Displaying a circular queue is just a little bit more complicated.

It has to print from the start to the end taking into account that more items may have been added to the head of the list

Priority Queue

A priority queue is a type of abstract data structure where each element has a priority. Instead of being processed in the order they were added (like a regular queue), elements are processed based on their priority.

Key Features:

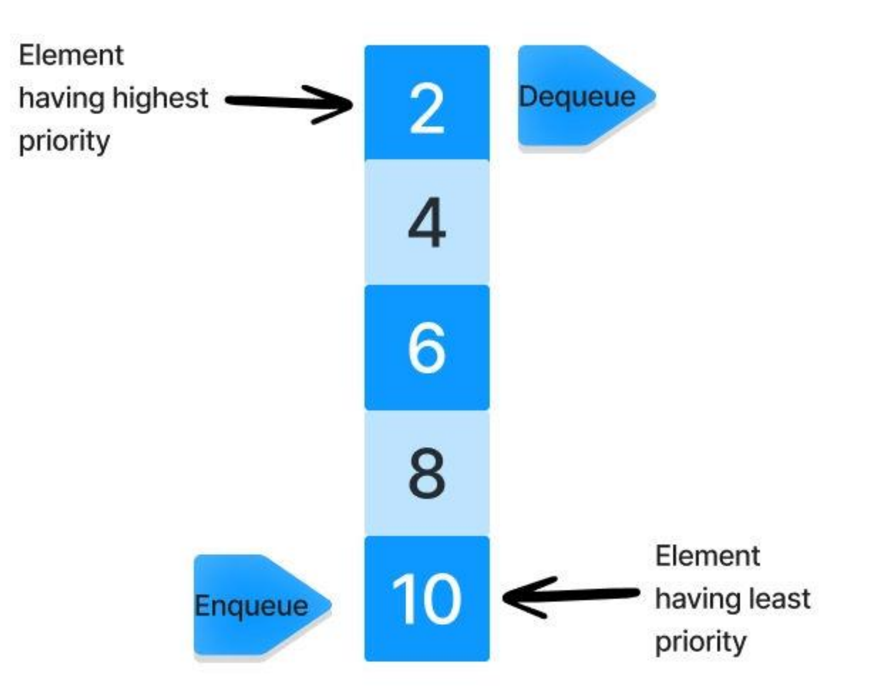
- Each item in the queue has:
- A value
- A priority level
- The item with the highest priority is removed first.
- If two items have the same priority, they are processed in FIFO (First In, First Out) order.

Operations:

- Insert (enqueue): Add an element with a priority
- Remove (dequeue): Remove the element with the highest priority
- May be implemented using:
- Arrays/lists (less efficient)
- Heaps (more efficient, e.g., binary heap)

Example Use Cases:

- Task scheduling (e.g., CPU processes)
- Dijkstra’s algorithm (for shortest path in graphs)
- Emergency room triage systems



Why use queues?

- Queues are often used **as buffers in programs**. They store data and objects in the order it arrives and then the program processes the objects in order.
- An example is in a **web browser constructing a full web page** to display. The browser will process the, usually small, html file immediately and store the other items in a queue as they arrive.. With a slow internet connection or a particularly complicated webpage you can often see this happening.
- Another example might be **when copying a large file from a hard disc drive to a usb stick**. The data will be read from the hard drive (possibly in chunks), **stored temporarily in a queue in RAM** and then written to the usb drive. It cannot be written directly because the read speed of the hard disc and the write speed of the usb will not be the same.