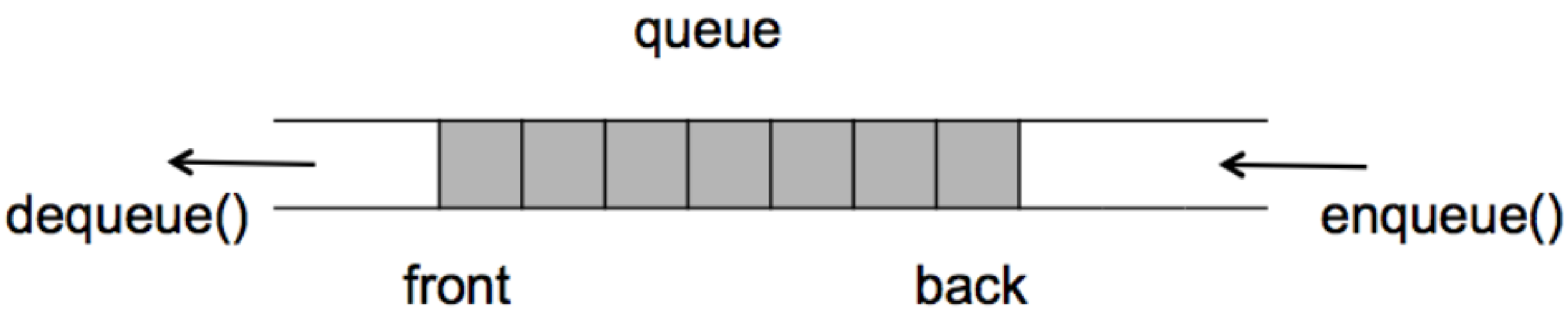
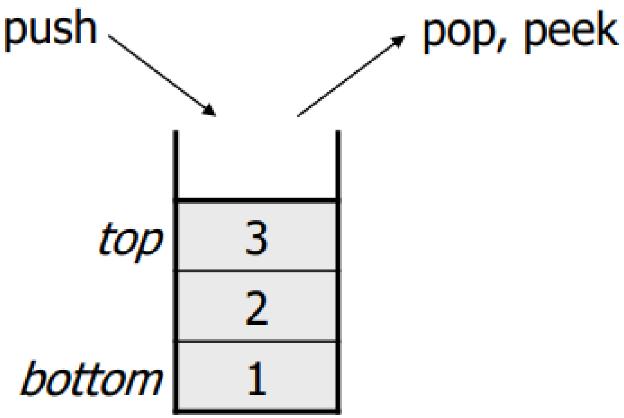
Queue

Queue – The first in fist out (FIFO)

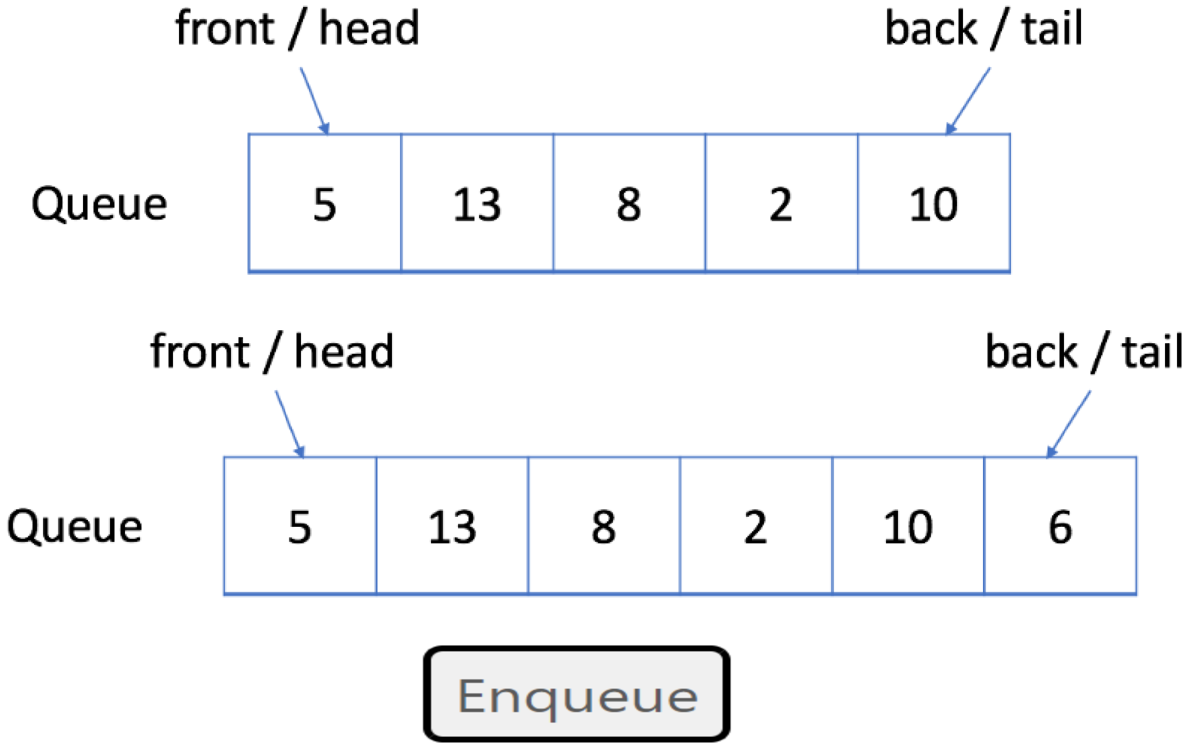
- A queue is a linear data structure that stores a collection of elements,
  - where the addition of new elements occurs at one end, called the "rear" or "tail“
  - the removal of elements occurs at the other end, called the "front" or "head".
- They can be implemented using arrays, linked lists, or other data structures,
- Like stack, often used in combination with other algorithms and data structures to solve **more complex problems**.

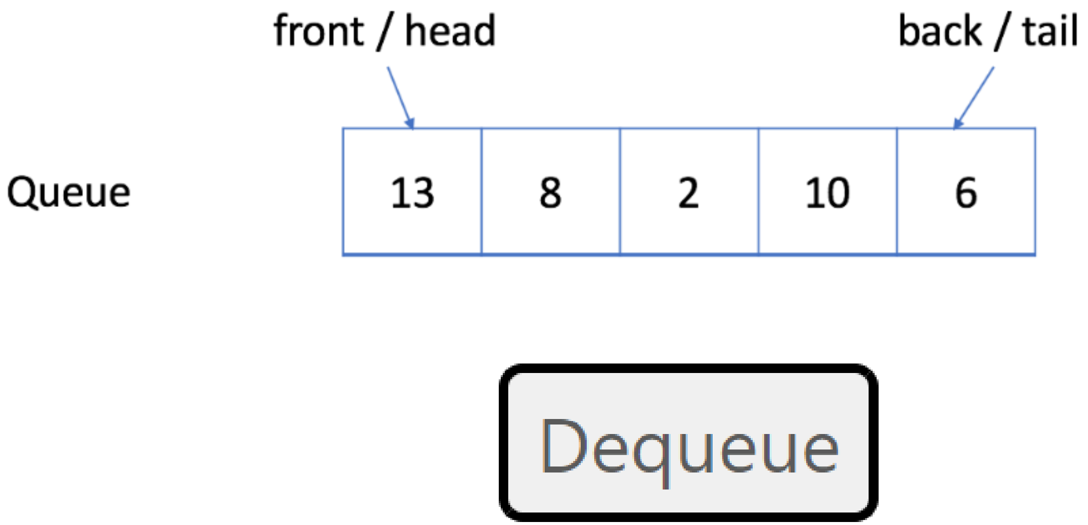




Queue – Operations

- ❑Enqueue: Add an element to the rear (tail) of the queue.
- ❑Dequeue: Remove the element at the front (head) of the queue.
- ❑Peek/Front: Retrieve the element at the front of the queue without removing it.
- ❑IsEmpty: Check if the queue is empty.
- ❑IsFull: Check if the queue is full (in case of a fixed-size queue).





Queue/Stack – Implementation

Array vs Dynamic array vs Linked list

Memory Allocation

- A dynamic array is similar to an array, but its size can be changed at runtime by allocating a new block of memory with a different size and copying the elements from the old block to the new one.

Access time

- Accessing an element in a dynamic array is fast (vs linked list), resizing the array can be slow.

Implementation Complexity

- Implementing a dynamic array is more complex than array, since it requires managing the allocation and deallocation of memory blocks.

Queue/Stack – Implementation

Array vs Dynamic array vs Linked list

- In their simplest form, arrays are not considered as ADTs, while dynamic arrays and linked lists can be considered as ADTs.
  - Dynamic arrays allow the user to add and remove elements dynamically, without having to manage the underlying memory allocation and deallocation.
  - The user only needs to know how to perform the operations, and not how they are implemented.
- Note: some programming languages may provide additional abstractions on top of arrays that make them ADTs.
  - For instance, the ArrayList class of Java provides methods to add, remove, and retrieve elements without exposing the underlying implementation details.

Queue – Array Implementation

```python
def display(self):
    if self.is_empty():
        print("Queue is empty")
    else:
        print("Queue: ", end='')
        for i in range(len(self.items)):
            print(self.items[i], end=' ')
        print()
```

```python
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.size() == 0

    def enqueue(self, item):
        self.items = self.items + [item]

    def dequeue(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        item = self.items[0]
        self.items = self.items[1:]
        return item

    def peek(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        return self.items[0]

    def size(self):
        count = 0
        for item in self.items:
            count += 1
        return count
```

Queue – Linked List

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Queue:
    def __init__(self):
        self.head = None
        self.tail = None
        self._size = 0

    def is_empty(self):
        return self.size() == 0

    def enqueue(self, item):
        new_node = Node(item)
        if self.is_empty():
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            self.tail = new_node
        self._size += 1
```

```python
def dequeue(self):
    if self.is_empty():
        raise Exception("Queue is empty")
    item = self.head.data
    self.head = self.head.next
    self._size -= 1
    return item

def peek(self):
    if self.is_empty():
        raise Exception("Queue is empty")
    return self.head.data

def size(self):
    return self._size
```

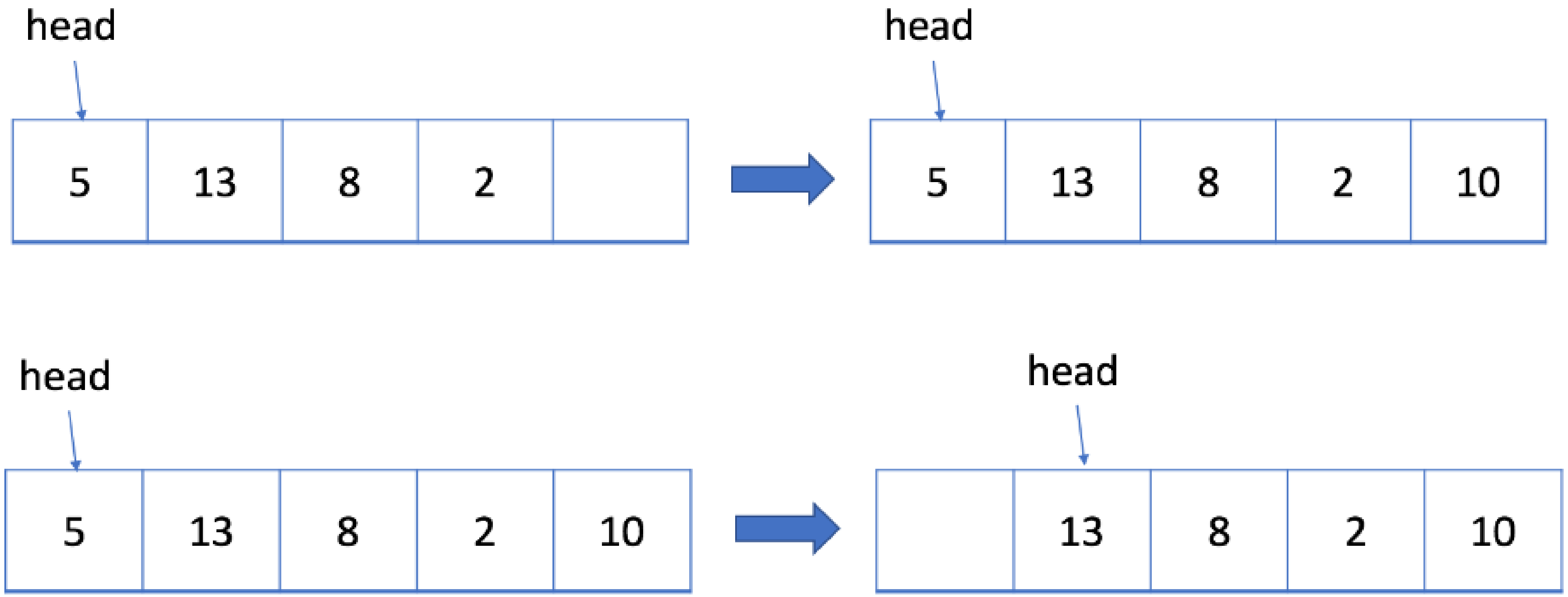
```python
def display(self):
    if self.is_empty():
        print("Queue is empty")
    else:
        current = self.head
        print("Queue: ", end='')
        while current is not None:
            print(current.data, end=' ')
            current = current.next
        print()
```

```python
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
q.display()
print(q.peek())
print(q.dequeue())
print(q.size())
```

```
Queue: 1 2 3
1
1
2
```

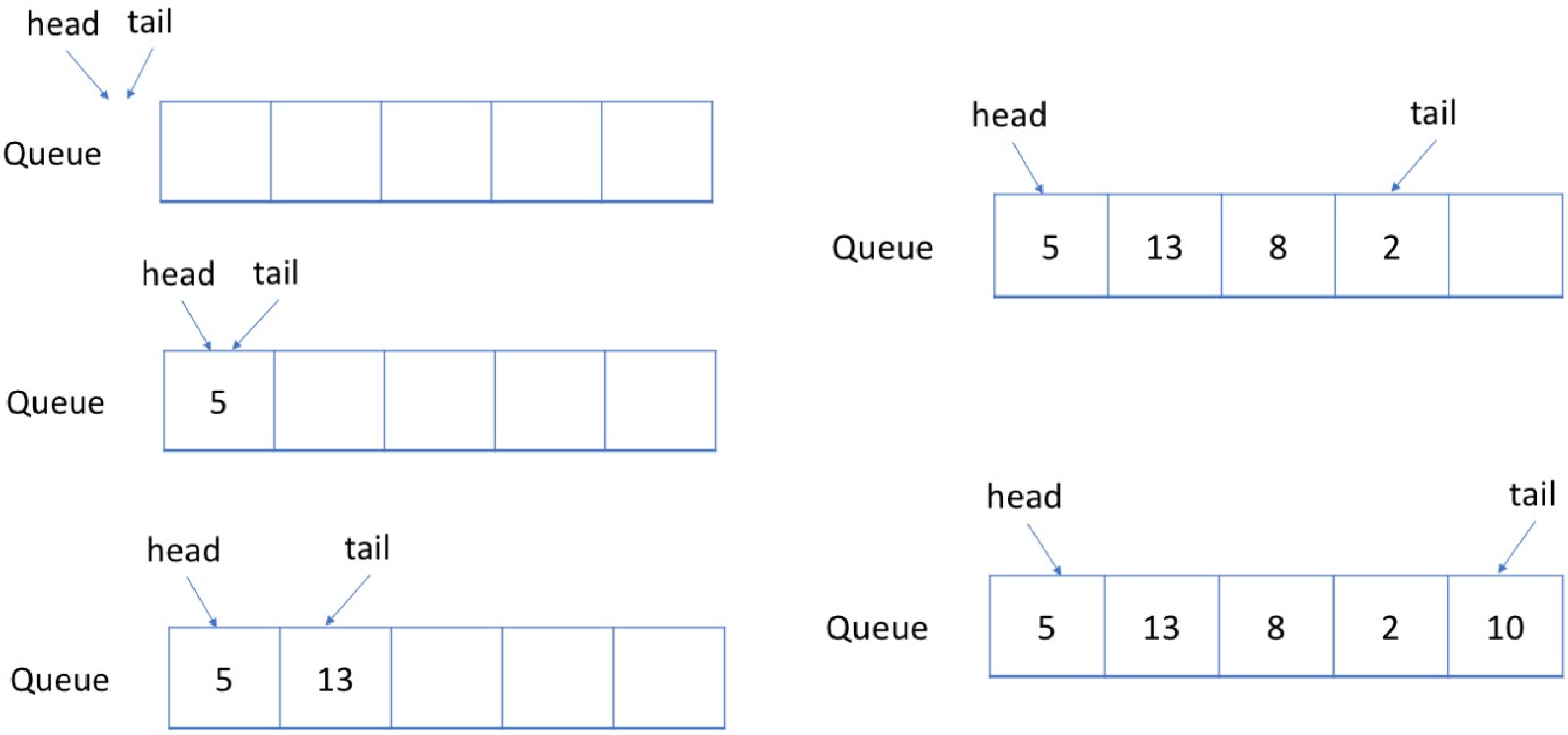
Circular Queue – Why?

- we are only able to allocate an array whose maximum length is 5

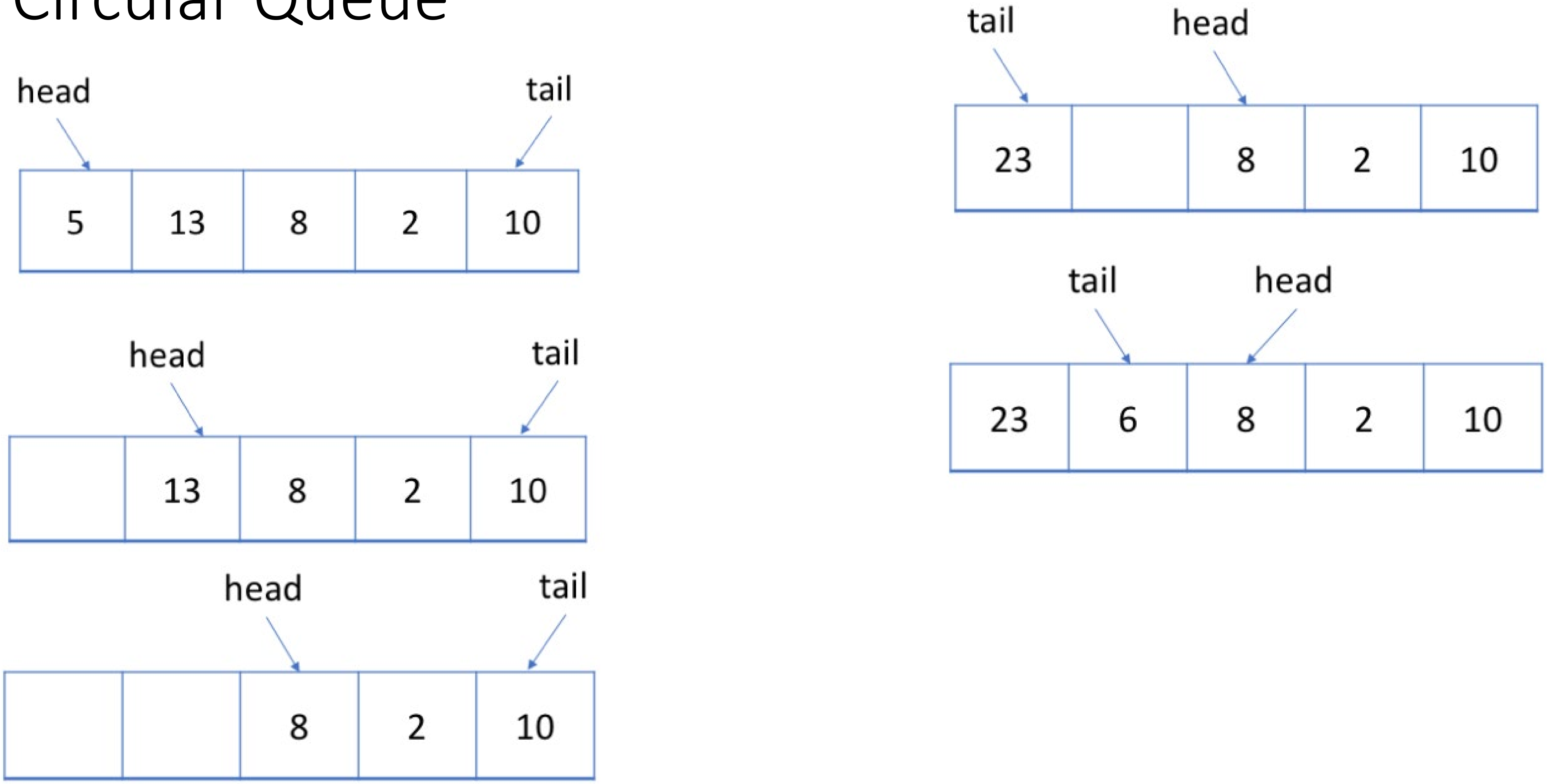


Accept one more element? Can we do that?

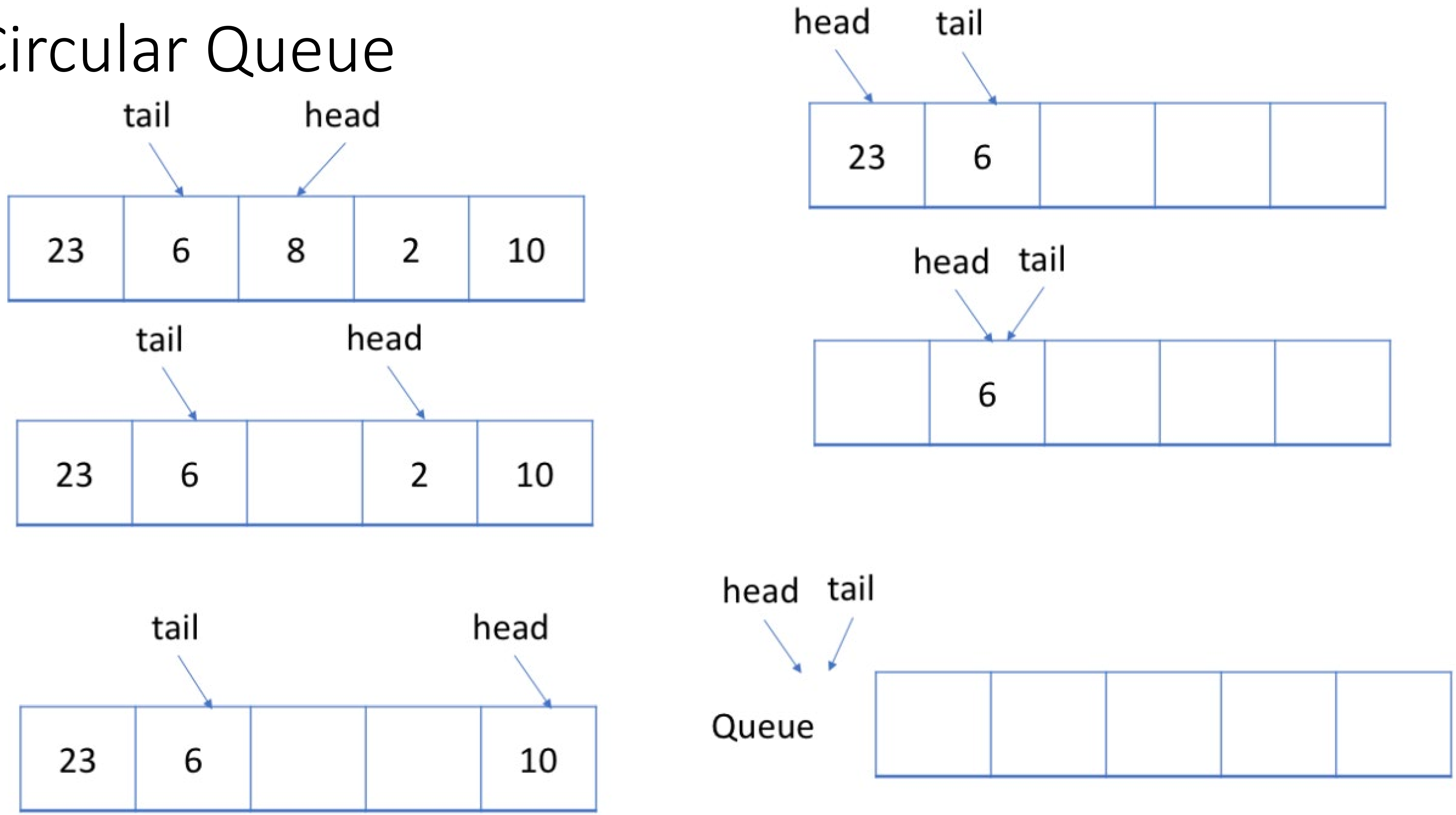
Circular Queue



Circular Queue



Circular Queue



Circular Queue

- The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle
- and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

- In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue.
- Using the circular queue, we can use the space to store new values.

Priority Queue

- A priority queue is a special type of queue in which each element is associated with a priority value.
- Elements are served on the basis of their priority. That is, higher priority elements are served first.
  - if elements with the same priority occur, they are served according to their order in the queue.

In a queue, the first-in-first-out rule is implemented whereas, in a priority queue, the values are removed on the basis of priority. The element with the highest priority is removed first.

Priority Queue – Implementation

- Unsorted List Implementation
  - stores the elements in an unsorted list and performs a linear search (i.e. O(n)) to find the element with the highest priority when dequeuing.

- Sorted List Implementation
  - stores the elements in a sorted list based on their priorities
  - When an element is enqueued, it is inserted into the appropriate position in the list using a binary search, which has a time complexity of *O(log n).*
  - When an element is dequeued, the element with the highest priority is removed from the end of the list, which has a time complexity of O(1).
  - inserting an element into the middle of the list requires shifting all the higher-priority elements one position to the right, which has a time complexity of O(n).

Priority Queue – Implementation

- Both of the implementations mentioned in the last slide are less efficient than a heap-based implementation for large numbers of elements
- they can be useful for small datasets or for situations where the priority queue is only used occasionally and efficiency is not a critical concern.

Priority Queue – Implementation

Operations	peek	insert	delete
Linked List	`O(1)`	`O(n)`	`O(1)`
Binary Heap	`O(1)`	`O(log n)`	`O(log n)`
Binary Search Tree	`O(1)`	`O(log n)`	`O(log n)`

Palindrome cheeker Using stack and queue

1. Find the mid as mid = len / 2.
2. Push all the elements till mid into the stack
3. In case of a odd length string, ignore the middle character.
4. Till the end of the string, keep popping elements from the stack and compare them with the current character
5. If there is a mismatch then the string is not a palindrome. If all the elements match then the string is a palindrome.

Palindrome cheeker Using stack and queue

```python
def check_palindrome(string):
    stack = []
    for char in string:
        stack.append(char)
    new_string = ''
    while len(stack) > 0:
        new_string += stack.pop()
    return new_string == string

check_palindrome("ababa")
```

Palindrome cheeker Using stack and queue

```python
def check_palindrome(string):
    queue = []
    for char in string:
        queue.append(char)
    new_string = ''
    while len(queue) > 0:
        new_string += queue.pop(0)
    return new_string == string
```

The Josephus problem

- The Josephus problem is named after Flavius Josephus, a Jewish historian who lived in the first century AD.
- The problem has been studied by mathematicians and computer scientists for many years.
- Has applications in a wide range of fields, including cryptography, Game Theory, and Computer Science.
- Eliminate each other in a circle, with every third (i.e. k^{th}) soldier being killed

The Josephus problem

- Eliminate each other in a circle, with every third (i.e. k^{th}) of n soldiers being killed

The Josephus problem – Applications

- **Memory allocation:** The Josephus problem can be used to allocate memory blocks in a circular buffer, where every kth block is allocated.
- **Processor scheduling:** can be used to schedule processes in a round-robin fashion, where every kth process is executed.
- **Data encryption:** can be used to encrypt data by applying a permutation to the data based on the Josephus sequence.
- **Parallel computing:** can be used to assign tasks to a set of processors in a circular fashion, where every kth processor is assigned a task.
- **Network routing:** can be used to route data packets in a circular network, where every kth node is used to forward the packet.

The Josephus problem – Implementation

- In the Josephus problem, a circular queue is a natural fit, since we are removing every k^{th} person from a circle of people.
- By treating the people as a circular queue, we can easily simulate the process of removing people from the circle by rotating the queue by k-1 positions and then dequeuing the person at the front of the queue.

Front 3rd tail

1	2	3	4	5	6	7
4	5	6	7	1	2	
7	1	2	4	5		
4	5	7	1			
1	4	5				
1	4	—				

~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~1~~ 3 5

1 2 3 4 5 6 7