Data Structures & Algorithms

Week 8

Queues

Lecturer: Dr. Msagha J Mbogholi, PhD

Flashback from Lesson 7

- Stacks are linear collections in which access is completely restricted to just one end, called the **top**. Conversely, the bottom of the stack is also known as its base.
- The operations on a stack are push, pop and peek (to see the element at the top of the stack).
- The two types of stacks are memory stacks and register stacks.
- Stacks use the LIFO (Last In, First Out) method where the element at the bottom never gets to be used unless there's no element on top of it.
- Stacks are implemented in arrays, lists and linked lists.
- Stacks are used in real life in areas such as call center systems, undo and Redo mechanism in text editors; the history of a web browser is stored in the form of a stack.

Content

- Queue ADT
- Implementation
- Priority queues
- Advantages and disadvantages
- Applications

Queue ADT

1.1 Introduction

- When I was in high school, I was a departmental prefect. It was a big deal back then since there were only three of us in the whole school population. In terms of seniority only the school captain was senior to us.
- Our duties together with other school prefects included ensuring orderliness in the student body, and punishing those who did not abide by the rules (to a certain limit). Disciplinary cases of higher order (they were defined) were referred to the school administration.
- We had a cafeteria system in the dining hall; this meant that students would queue in an orderly fashion for food. Of course prefects did not have to queue; they were a priority group and they would just either walk to the front of the queue and get served, or some would walk directly into the kitchen to get served.

1.1 Introduction

- Prefects would be mostly identified as they wore a different shirt color from the rest of the school population; however, with time the population knew who the prefects were, regardless of whether they were wearing the different color shirt or not.
- The cafeteria system (queue) was monitored by the prefect on duty and it was his role to ensure that the queue proceeded in an orderly manner, and that latecomers would not cut to the front before others (happened a lot between senior and junior students).
- The advantage of this system is that it ensured orderliness, and students who came early got to be served first before those who came later. Take note that the system worked for all students; only difference being prefects who got preferential treatment in terms of being given priority.

1.1 Introduction

- Fast forward. ATMs landed in my country when I was barely out of school. They were brought in due to ease congestion in banking halls, which had truly gotten out of hand. I remember Standard Chartered bank were the first bank to introduce ATMs, since we would all just go to watch people use them. Nonetheless it took a long time for customers to adapt to this change and use them as we all do today.
- Nowadays we only go into the banking hall for services not available at the ATM. The issue then was that customers had to queue for services. Unfortunately there were no prefects in banking halls like in my school days; so customers would find all unfair ways to get served before others. A common ploy was to walk into the banking hall and go somewhere near the front of the queue and declare you had been there but had stepped out to pick a call (or see the customer care, or fill in a banking slip, and so on) and fix yourself in front of others.
- This led to many confrontations between customers, and physical encounters weren't uncommon. To ameliorate this the bank officials decided to have a member of staff (or three) monitor the lines throughout the day.

1.1 Introduction

- This did not solve the problem. Some time later they introduced benches in the banking hall so that those who couldn't stand for long periods could sit and wait as the queue proceeded, and join the line nearer to the front when their turn came. Inevitably, this was also abused. Is this human nature I wonder?
- People would sit on the benches and move to the front of the queue, claiming that was their spot even when it wasn't. Goodness me. Fast forward again.
- A few years back, banks came up with what might be the best solution yet; the ticketing systems, or what we call queue management systems (QMS). A customer walks into the bank goes to the machine, chooses a service, gets a ticket for it, and sits and waits for their ticket number to be called. When your ticket number is called the announcement also tells you which counter to go to be served. Again priority is available as most banks have an executive accounts (or what they call prestige accounts) and these are served faster than the rest of the customers.

1.1 Introduction

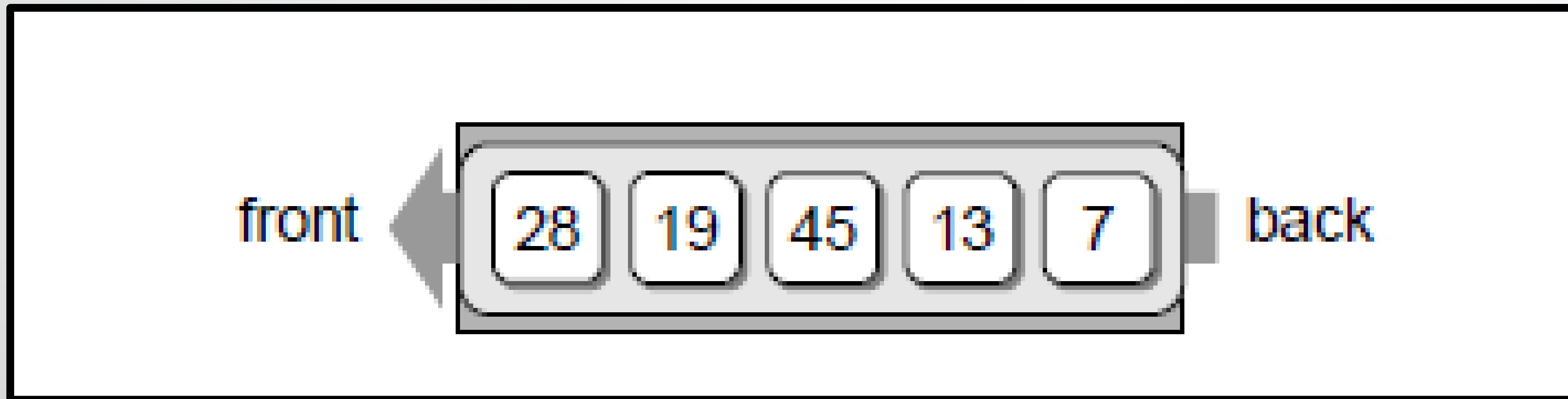
- However, hitherto, there are still banks who maintain a member of staff at the QMS ticket machine to help customers use it, and also to direct customers to start here before taking a seat awaiting service.
- What's the story behind all this introduction narrative, you may wonder?
- The concept of queues to bring about an orderly manner of service is documented from as early as the 19th century. There are different ways of queuing but the idea is the same; people get served in an orderly and timely fashion…it is not random.
- The concept of queues as can be seen, has been taken up in computing; it is one of our discrete structures and is the topic of this lesson. As shall be seen, queueing is also implemented using other data structures that have already been introduced in this course so far.

1.2 Queue ADT

- A queue is a specialized list with a limited number of operations in which items can only be added to one end and removed from the other.
- Karumanchi (2017) defines a queue as "… an ordered list in which insertions are done at one end (*rear*) and deletions are done at other end (*front*). The first element to be inserted is the first one to be deleted. Hence, it is called First in First out (FIFO) or Last in Last out (LILO) list."
- Figure 1 illustrates an abstract view of a queue. New items are inserted into a queue at the back while existing items are removed from the front. Note that items are removed from the queue only from the front and removed only from the back; this means there is no access to elements in between till they get to the front of the queue.
- Figure 2 shows the definition of the queue as an ADT, while table 1 shows the methods associated with the queue interface.

Figure 1

Abstract view of queue



(From Necaise, 2011)

Figure 2

Queue ADT definition

Define	Queue ADT

A *queue* is a data structure that a linear collection of items in which access is restricted to a first-in first-out basis. New items are inserted at the back and existing items are removed from the front. The items are maintained in the order in which they are added to the structure.

- `Queue()`: Creates a new empty queue, which is a queue containing no items.
- `isEmpty()`: Returns a boolean value indicating whether the queue is empty.
- *`length`* `()`: Returns the number of items currently in the queue.
- `enqueue( item )`: Adds the given item to the back of the queue.
- `dequeue()`: Removes and returns the front item from the queue. An item cannot be dequeued from an empty queue.

(From Necaise, 2011)

Table 1

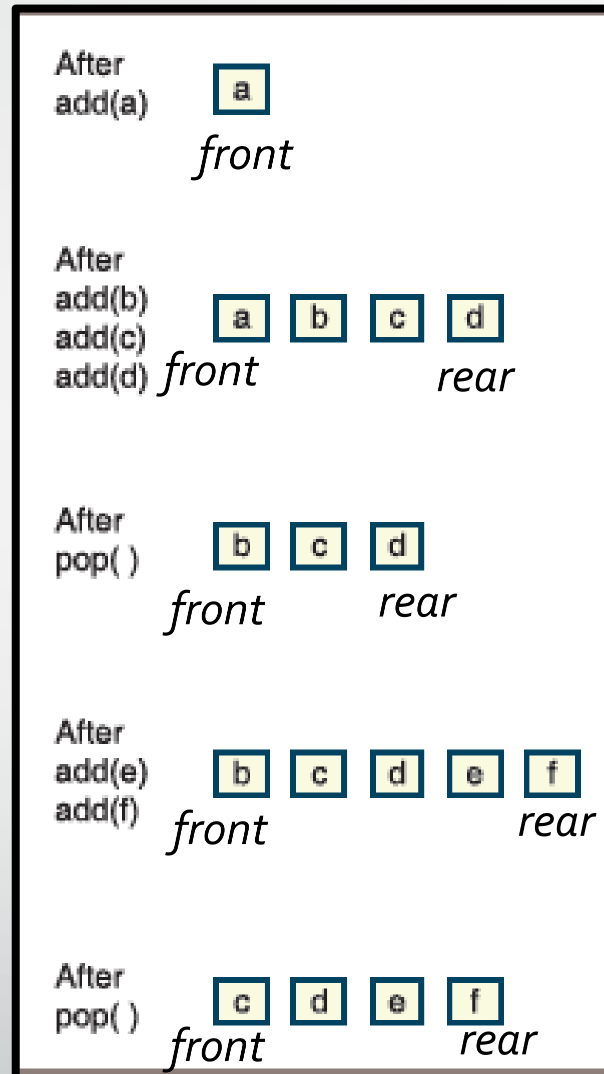
Queue methods

Queue Method	What It Does
`q.isEmpty()`	Returns `True` if `q` is empty or `False` otherwise.
`__len__(q)`	Same as `len(q)`. Returns the number of items in `q`.
`__str__(q)`	Same as `str(q)`. Returns the string representation of `q`.
`q.__iter__()`	Same as `iter(q)`, or `for item in q:`. Visits each item in `q`, from front to rear.
`q.__contains__(item)`	Same as `item in q`. Returns `True` if item is in `q` or `False` otherwise.
`q1__add__(q2)`	Same as `q1 + q2`. Returns a new queue containing the items in `q1` followed by the items in `q2`.
`q.__eq__(anyObject)`	Same as `q == anyObject`. Returns `True` if `q` equals `anyObject` or `False` otherwise. Two queues are equal if the items at corresponding positions are equal.
`q.clear()`	Makes `q` become empty.
`q.peek()`	Returns the item at the front of `q`. *Precondition*: `q` must not be empty; raises a `KeyError` if the queue is empty.
`q.add(item)`	Adds `item` to the rear of `q`.
`q.pop()`	Removes and returns the item at the front of `q`. *Precondition*: `q` must not be empty; raises a `KeyError` if the queue is empty.

(From Lambert, 2019)

Figure 3

Lifetime of a queue



(Adapted from Lambert, 2019)

1.2 Queue ADT

- Figure 3 shows us the operations of the queue.
- Initially the item a is added to the queue.
- In the second step b, c, and d are added to the queue. Note that items are added from the rear of the queue (labelled 'rear' in the figure). This is similar to you joining an existing queue; you join from the back, right?
- In the third step a is removed from the queue via a pop. Note that removal from the queue is only from the front of the queue (labelled 'front' in the figure). This is similar to you being served next in the cafeteria analogy described in the introduction of this lesson. You are served and move out of the queue (in computer language this is implemented using the pop method).
- In the fourth step three elements are added namely, e and f.
- In the last step (fifth), b is removed (popped) from the front of the queue.
- These operations are captured in table 2 using the appropriate Python methods. Finally table 3 captures the performance of queue operations using complexity analysis.

Table 2

Queue operations using Python

Operation	State of the Queue After the Operation	Value Returned	Comment
q = <Queue Type>()			Initially, the queue is empty.
q.add(a)	a		The queue contains the single item **a**.
q.add(b)	a b		**a** is at the front of the queue and **b** is at the rear.
q.add(c)	a b c		**c** is added at the rear.
q.isEmpty()	a b c	False	The queue is not empty.
len(q)	a b c	3	The queue contains three items.
q.peek()	a b c	a	Return the front item on the queue without removing it.
q.pop()	b c	a	Remove the front item from the queue and return it. **b** is now the front item.
q.pop()	c	b	Remove and return **b**.
q.pop()		c	Remove and return **c**.
q.isEmpty()		True	The queue is empty.
q.peek()		exception	Peeking at an empty queue throws an exception.
q.pop()		Exception	Trying to pop an empty queue throws an exception.
q.add(d)	d		**d** is the front item.

(From Lambert, 2019)

1.2 Queue ADT

- Using the methods described in figure 2 we can rewrite the code for table 2 as follows:
- Q = Queue()   #create the queue
- Q.enqueue(a)   #add the element a to the rear of the queue
- Q.enqueue(b)   #add the element b to the rear of the queue
- Q.enqueue(c)   #add the element c to the rear of the queue
- Q.enqueue(d)   #add the element d to the rear of the queue
- Q.dequeue(a)   #remove the element a from the front of the queue
- Q.enqueue (e)   #add the element e to the rear of the queue
- Q.enqueue (f)   #add the element f to the rear of the queue
- Q.dequeue (b)   #remove the element b from the front of the queue
- These methods will return the same results.

Table 3

Complexity analysis of queue operations

Operations	Complexity
Enqueue(insertion)	O(1)
Deque(deletion)	O(1)
Front(Get front)	O(1)
Rear(Get Rear)	O(1)
IsFull(Check queue is full or not)	O(1)

(From *Introduction and Array Implementation of Queue*, 2014)

Part 2
Implementations

2.1 Introduction

- Similar to what was discussed in lesson 7, queues can be implemented in three data structures:
  - Lists
  - Linked lists
  - Arrays
- The implementations (and accompanying code) are found in many literature; however, these are discussed in this part of the lesson using select sources.

2.2 Using lists

- Using a list is the easiest way to implement the queue ADT. In this manner we can add items to the rear of the queue, and similarly remove items from the front.
- Between the ADT definition in figure 1, and the methods defined in table 1 we can define a class for our list that can perform all the necessary operations; to add and remove items from the list.
- The constructor only needs to create an empty list, and from there items can be added or removed from the list. The necessity for this is due to the fact that if we attempt to remove items from an empty list this will result in an error called underflow (dequeuing an empty list); similarly attempting to add an item to an already full list will result in an overflow error (attempt to enqueue a full list).
- Let us examine the code for the class illustrated in figure 4. We then apply this to an example.

Figure 4

Queue ADT

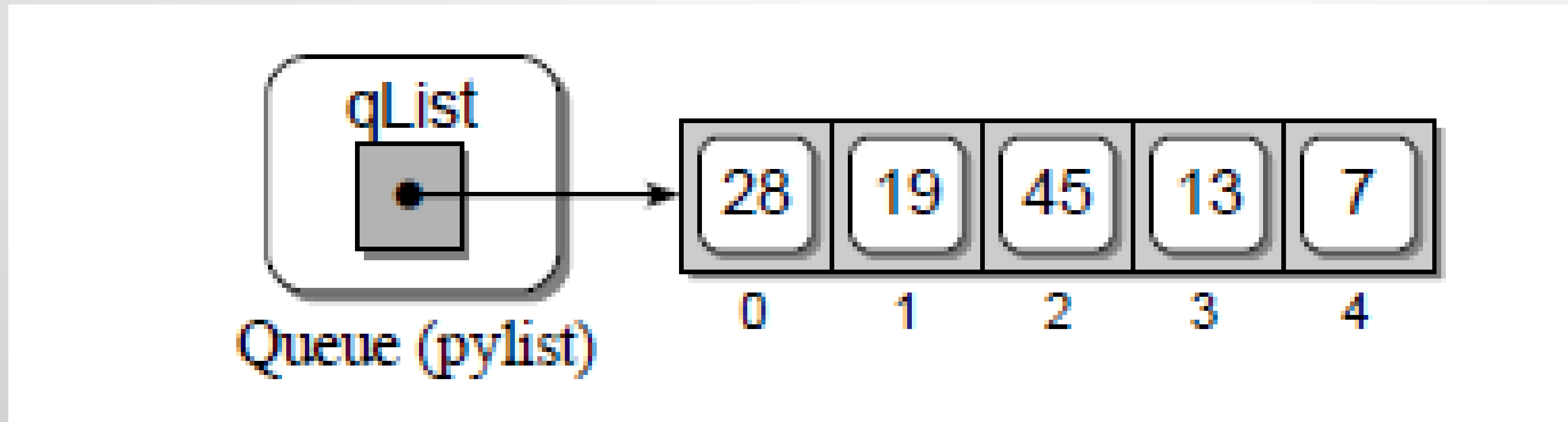
using Python list

```
1  # Implementation of the Queue ADT using a Python list.
2  class Queue :
3     # Creates an empty queue.
4    def __init__( self ):
5      self._qList = list()
6
7     # Returns True if the queue is empty.
8    def isEmpty( self ):
9      return len( self ) == 0
10
11    # Returns the number of items in the queue.
12   def __len__( self ):
13     return len( self._qList )
14
15    # Adds the given item to the queue.
16   def enqueue( self, item ):
17     self._qList.append( item )
18
19    # Removes and returns the first item in the queue.
20   def dequeue( self ):
21     assert not self.isEmpty(), "Cannot dequeue from an empty queue."
22     return self._qList.pop( 0 )
```

(From Necaise, 2011)

Figure 5

Queue ADT implemented in list



(From Necaise, 2011)

2.2 Using lists

- Figure 5 shows us the implementation of the class on an object qList. Let us show the code for creating the list, checking if it is empty and adding the first two items; the remaining items are left as an exercise for you…just follow the code I have used here ☺

```
qList = Queue()  #create the object qList
qList isEmpty()  #confirm the list is empty
qList.enqueue(28) #first item on the list
qList.enqueue(19)  #second item on the list
……. Enqueue the remaining items.
qList.len()    #return the length of the queue (should return 5 for number of items)
qList.dequeue()#remove the item at the front of the list (28 in this case)
qList.len()    #should now return 4, since 1 item has been removed from the queue
```

2.2.1 Performance

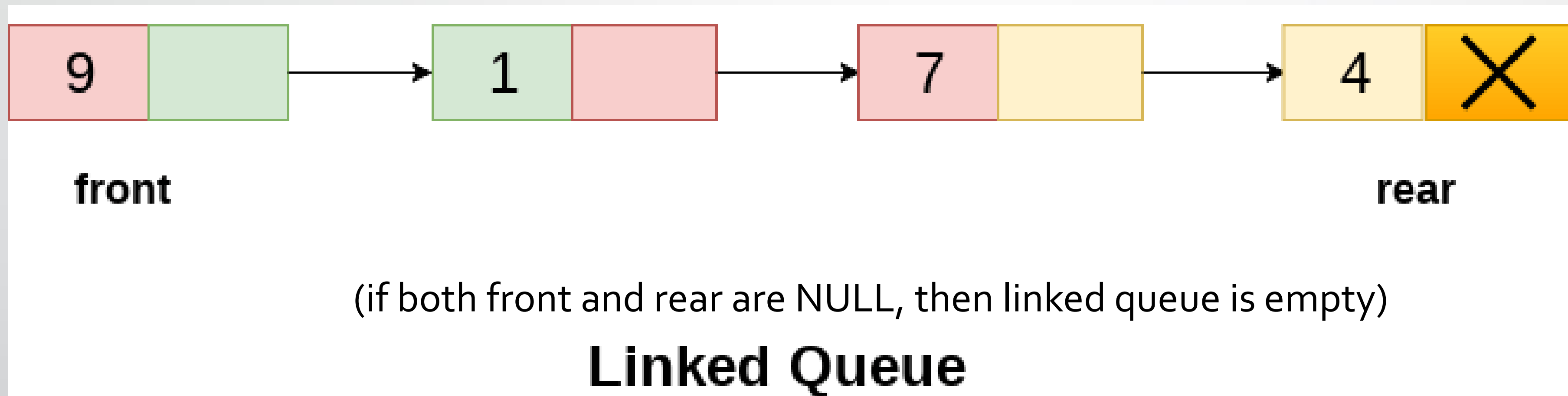
- Examining the performance using complexity reveals the following (Necaise, 2011):
- The size and empty condition operations only require O(1) time.
- The enqueue operation requires O(n) time in the worst case since the list may need to expand to accommodate the new item.
- When used in sequence, the enqueue operation has an amortized cost of O(1).
- The dequeue operation also requires O(n) time since the underlying array used to implement the Python list may need to shrink when an item is removed.
- In addition, when an item is removed from the front of the list, the following items have to be shifted forward, which requires linear time no matter if an expansion occurs or not.

2.3 Using Linked lists

- The linked list implementation is used for large scale applications where queues are implemented.
- The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).
- We learnt in earlier lessons that in a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.
- In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.
- Insertion is performed at the rear and deletions are performed at front end; therefore, if front and rear both are NULL, it indicates that the queue is empty.
- The linked representation of queue is shown in the following figure 6.

Figure 6

Linked list



(Adapted from *Linked List Implementation of Queue - Javatpoint*, 2011)

2.3 Using Linked lists

- Necaise (2011) also notes the following: "A major disadvantage in using a Python list to implement the Queue ADT is the expense of the enqueue and dequeue operations… A better solution is to use a linked list consisting of both head and tail references. Adding the tail reference allows for quick append operations that otherwise would require a complete traversal to find the end of the list."
- Necaise (2011) also provides the Python implementation code for the linked list queue in figure 8, while adding "Remember, the individual nodes in the list contain the individual items in the queue. When dequeuing an item, we must unlink the node from the list but return the item stored in that node and not the node itself." this is a very important point to note.
- Table 4 shows the performance linked list queue in terms of complexity analysis

Figure 8 (a)

Linked queue implementation

(From Necaise, 2011)

```
1  # Implementation of the Queue ADT using a linked list.
2  class Queue :
3      # Creates an empty queue.
4    def __init__( self ):
5      self._qhead = None
6      self._qtail = None
7      self._count = 0
8
9     # Returns True if the queue is empty.
10   def isEmpty( self ):
11     return self._qhead is None
12
13    # Returns the number of items in the queue.
14   def __len__( self ):
15     return self._count
16
17    # Adds the given item to the queue.
18   def enqueue( self, item ):
19     node = _QueueNode( item )
20     if self.isEmpty() :
21       self._qhead = node
22     else :
23       self._qtail.next = node
24
25     self._qtail = node
26     self._count += 1
27
28    # Removes and returns the first item in the queue.
29   def dequeue( self ):
30     assert not self.isEmpty(), "Cannot dequeue from an empty queue."
31     node = self._qhead
32     if self._qhead is self._qtail :
33       self._qtail = None
```

Figure 8 (b)

Linked queue implementation

```
34
35      self._qhead = self._qhead.next
36      self._count -= 1
37      return node.item
38
39  # Private storage class for creating the linked list nodes.
40  class _QueueNode( object ):
41    def __init__( self, item ):
42      self.item = item
43      self.next = None
```

(From Necaise, 2011)

Table 4

Linked queue complexity analysis

Space Complexity (for n EnQueue operations)	O(n)
Time Complexity of EnQueue()	O(1) (Average)
Time Complexity of DeQueue()	O(1)
Time Complexity of IsEmptyQueue()	O(1)
Time Complexity of DeleteQueue()	O(1)

(From Karumanchi, 2017)

2.4 Using array

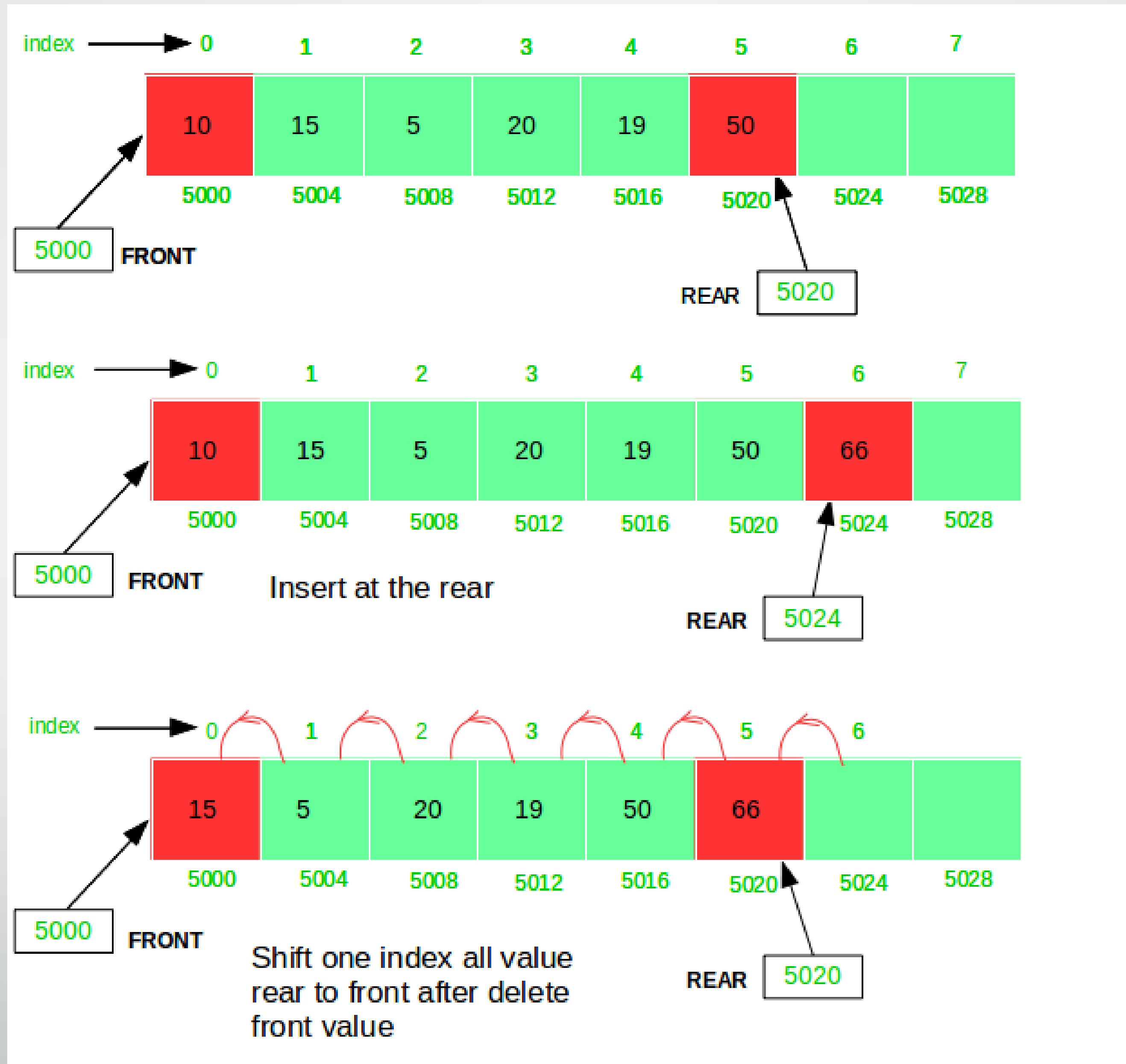
- When we wish to implement a queue using an array we take into consideration all the characteristics of the array discussed in lesson 4 of this course.
- In this section we consider two types of arrays:
  - The standard array
  - The circular array
- We discuss both and show how they are implemented in Python.

2.4.1 Using standard array

- (Rajput, 2018) describes the process as follows:
- Steps:
- Create an array arr of size **n** and
- take two variables **front** and **rear** both of which will be initialized to 0 which means the queue is currently empty.
- Element
  - rear is the index up to which the elements are stored in the array and
  - front is the index of the first element of the array.
- The available operations are the same ones discussed earlier in this lesson: enqueue, dequeue, front (to go to the front of the queue), and display (print the contents of the queue).
- Figure 9 shows these operations for ease of visualization.

Figure 9

Queue operations on array



(From Rajput, 2018)

2.4.1 Using standard array

- Rajput (2018) explains the code used in the implementation as follows:
- We are initializing front and rear as 0, but in general we have to initialize it with -1.
- If we assign rear as 0, rear will always point to next block of the end element, in fact , rear should point the index of last element, eg. When we insert element in queue , it will add in the end i.e. after the current rear and then point the rear to the new element ,
  According to the following code:
  IN the first dry run, front=rear = 0;
- In void queueEnqueue(int data), else part will be executed, so arr[rear] =data;// rear =0, rear pointing to the latest element
- rear++; //now rear = 1, rear pointing to the next block after end element not the end element
- //that's against the original definition of rear.
- The code itself follows:

```
# Python3 program to implement
# a queue using an array

class Queue:

    # To initialize the object.
    def __init__(self, c):

        self.queue = []
        self.front = self.rear = 0
        self.capacity = c
```

```
    # Function to insert an element
    # at the rear of the queue
    def queueEnqueue(self, data):

        # Check queue is full or not
        if(self.capacity == self.rear):
            print("\nQueue is full")

        # Insert element at the rear
        else:
            self.queue.append(data)
            self.rear += 1
```

```
# Function to delete an element
  # from the front of the queue
  def queueDequeue(self):

        # If queue is empty
        if(self.front == self.rear):
              print("Queue is empty")

        # Pop the front element from list
        else:
              x = self.queue.pop(0)
              self.rear -= 1
```

```
# Function to print queue elements
  def queueDisplay(self):

        if(self.front == self.rear):
              print("\nQueue is Empty")

        # Traverse front to rear to
        # print elements
        for i in self.queue:
              print(i, "<--", end='')
```

```
# Print front of queue
def queueFront(self):

        if(self.front == self.rear):
                print("\nQueue is Empty")

        print("\nFront Element is:",
                self.queue[self.front])
# Driver code
if __name__ == '__main__':

    # Create a new queue of
    # capacity 4
    q = Queue(4)

    # Print queue elements
    q.queueDisplay()

    # Inserting elements in the queue
    q.queueEnqueue(20)
    q.queueEnqueue(30)
    q.queueEnqueue(40)
    q.queueEnqueue(50)

    # Print queue elements
    q.queueDisplay()
```

- # Insert element in queue
- q.queueEnqueue(60)

- # Print queue elements
- q.queueDisplay()

- q.queueDequeue()
- q.queueDequeue()
- print("\n\nafter two node deletion\n")

- # Print queue elements
- q.queueDisplay()

- # Print front of queue
- q.queueFront()

- # This code is contributed courtesy Amit Mangal

2.4.1 Using standard array

- The code output is as follows:
- Queue is Empty
- 20 <-- 30 <-- 40 <-- 50 <--
- Queue is full
- 20 <-- 30 <-- 40 <-- 50 <--
- after two node deletion
- 40 <-- 50 <--
- Front Element is: 40

2.4.1 Using standard array

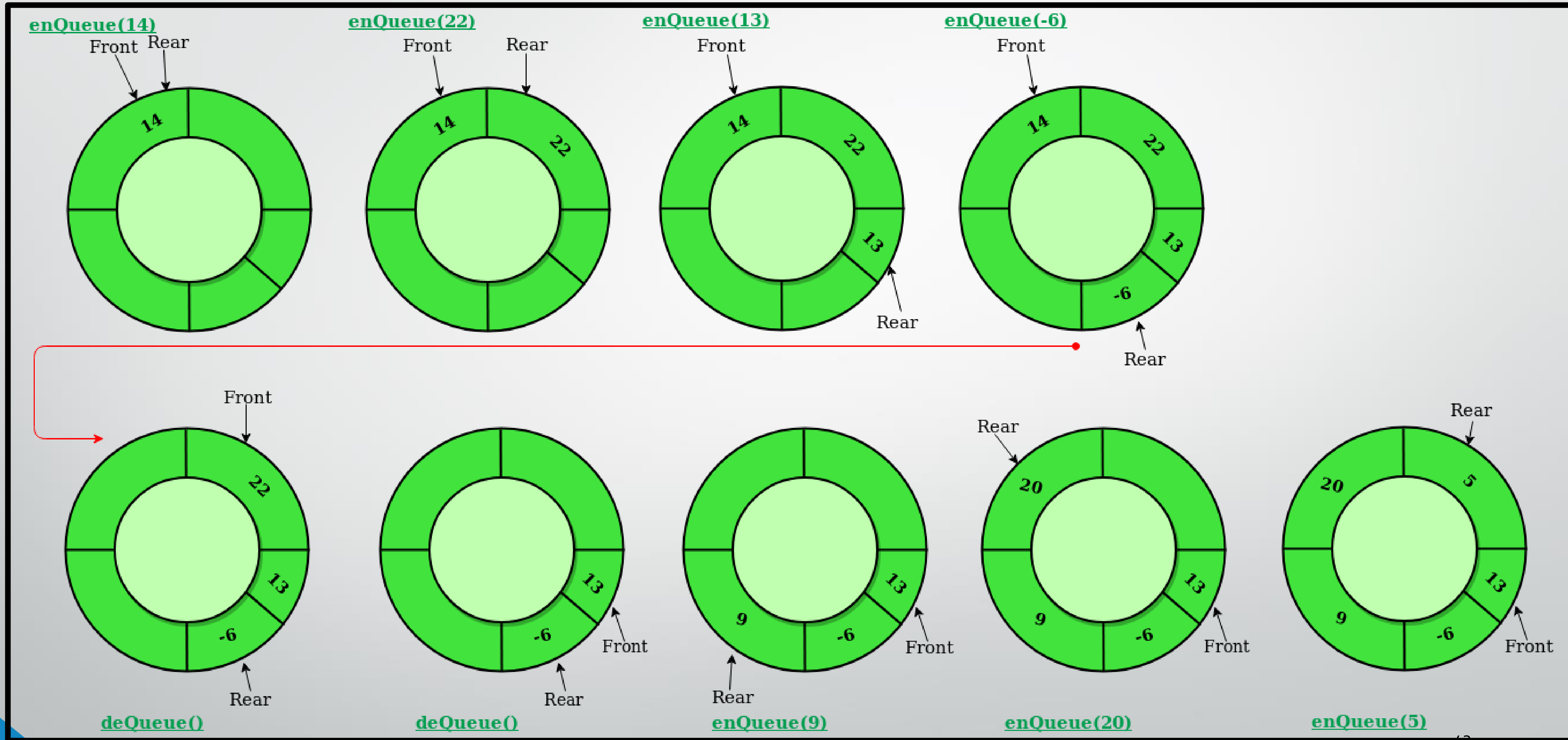
- The complexity is as follows:
- For Enqueue (element insertion in the queue): O(1) since we simply increment pointer and put value in array; and
- For Dequeue (element removing from the queue) : O(N) since we use for loop to implement that.

2.4.2 Using circular array

- A circular array is simply an array viewed as a circle instead of a line. (Gupta, 2017) describes it as "an extended version of a normal queue where the last element of the queue is connected to the first element of the queue forming a circle." It is also known as a ring buffer.
- Secondly the circular array also uses the FIFO principle. The circular array overcomes the handicap of the normal array; in the latter we can't add the next elements to the array even if the array is not full, like when there is space is at the front of the queue.
- The working of the circular array with its operations (queue implementation) are demonstrated in figure 10.

Figure 10

Circular array queue implementation



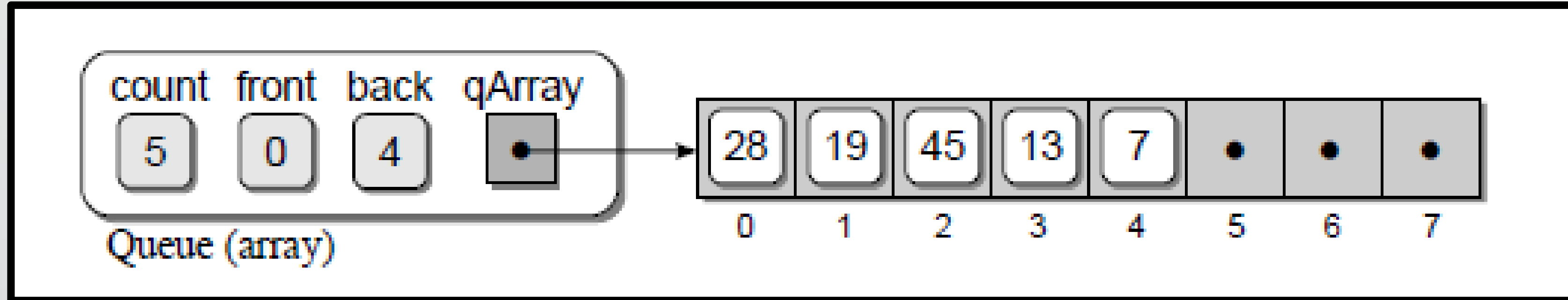
(From Gupta, 2017)

2.4.2 Using circular array

- Necaise (2011) provides the implementation code and explains as follows:
- The constructor creates an object containing four data fields, including the counter to keep track of the number of items in the queue, the two markers, and the array itself. A sample instance of the class is illustrated in Figure 11.
- For the circular queue, the array is created with maxSize elements as specified by the argument to the constructor. The two markers are initialized so the first item will be stored in element 0. This is achieved by setting _front to 0 and _back to the index of the last element in the array.
- When the first item is added, _back will wrap around to element 0 and the new value will be stored in that position. Figure 12 illustrates the circular array when first created by the constructor.
- Figure 13 shows the code for the implementation.

Figure 11

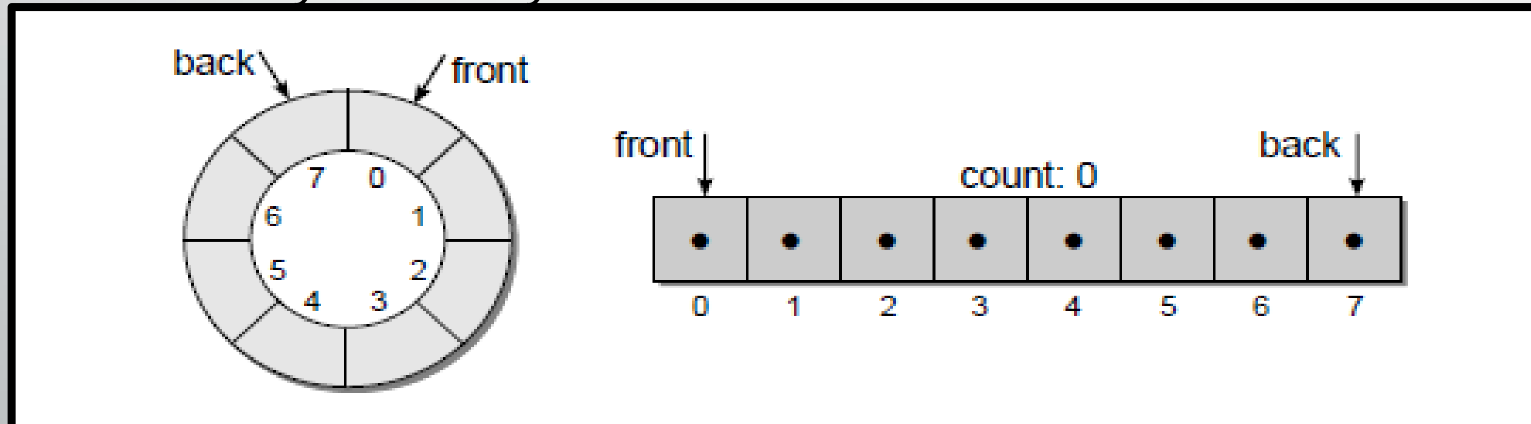
Sample instance of class



(From Necaise, 2011)

Figure 12

Circular array created by constructor



(From Necaise, 2011)

Figure 13

Implementation

of circular array queue

(From Necaise, 2011)

```
1  # Implementation of the Queue ADT using a circular array.
2  from array import Array
3
4  class Queue :
5      # Creates an empty queue.
6    def __init__( self, maxSize ) :
7      self._count = 0
8      self._front = 0
9      self._back = maxSize - 1
10     self._qArray = Array( maxSize )
11
12    # Returns True if the queue is empty.
13   def isEmpty( self ) :
14     return self._count == 0
15
16    # Returns True if the queue is full.
17   def isFull( self ) :
18     return self._count == len(self._qArray)
19
20    # Returns the number of items in the queue.
21   def __len__( self ) :
22     return self._count
23
24    # Adds the given item to the queue.
25   def enqueue( self, item ):
26     assert not self.isFull(), "Cannot enqueue to a full queue."
27     maxSize = len(self._qArray)
28     self._back = (self._back + 1) % maxSize
29     self._qArray[self._back] = item
30     self._count += 1
31
32    # Removes and returns the first item in the queue.
33   def dequeue( self ):
34     assert not self.isEmpty(), "Cannot dequeue from an empty queue."
35     item = self._qArray[ self._front ]
36     maxSize = len(self._qArray)
37     self._front = (self._front + 1) % maxSize
38     self._count -= 1
39     return item
```

Priority queues

3.1 Definition and operations

- Necaise (2011) defines a priority queue together with its operations as follows: “
- A priority queue is a queue in which each item is assigned a priority and items with a higher priority are removed before those with a lower priority, irrespective of when they were added. Integer values are used for the priorities with a smaller integer value having a higher priority. A bounded priority queue restricts the priorities to the integer values between zero and a predefined upper limit whereas an unbounded priority queue places no limits on the range of priorities.”
- Methods include:”
- PriorityQueue(): Creates a new empty unbounded priority queue.
- BPriorityQueue(numLevels): Creates a new empty bounded priority queue with priority levels in the range from 0 to numLevels - 1.

isEmpty(): Returns a boolean value indicating whether the queue is empty.

3.1 Definition and operations

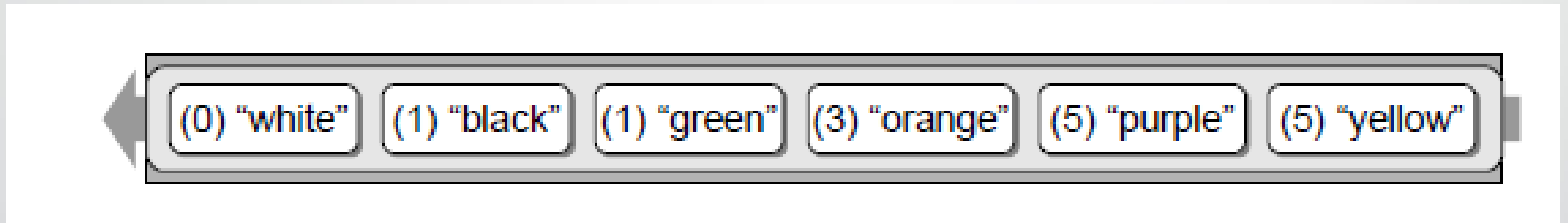
- length (): Returns the number of items currently in the queue.
- enqueue(item, priority): Adds the given item to the queue by inserting it in the proper position based on the given priority. The priority value must be within the legal range when using a bounded priority queue.
- dequeue(): Removes and returns the front item from the queue, which is the item with the highest priority. The associated priority value is discarded. If two items have the same priority, then those items are removed in a FIFO order. An item cannot be dequeued from an empty queue.
- Let us consider an example: we wish to enqueue a number of items into a priority queue. The priority queue is defined with six levels of priority with a range of [0 : : : 5]. The resulting queue is shown in Figure 14.

3.1 Definition and operations

- Q = BPriorityQueue(6)
- Q.enqueue(*"purple"*, 5)
- Q.enqueue(*"black"*, 1)
- Q.enqueue(*"orange"*, 3)
- Q.enqueue(*"white"*, 0)
- Q.enqueue(*"green"*, 1)
- Q.enqueue(*"yellow"*, 5)
- The first item to be removed will be the first item with the highest priority. In this case white has the highest priority so running the dequeue command will yield an output of (white, green, black, orange, purple, yellow). The Python implementation is left as an exercise for the learner. Simply follow the code for the queue class and remember that in this case you are reading the position together with the assigned priority.

Figure 14

Priority queue with priority levels



(From Necaise, 2011)

3.2 Data structure implementation

- Priority queue can be implemented using the following data structures:
  - Arrays
  - Linked list
  - Heap data structure
  - Binary search tree

Advantages and disadvantages

4.1 Advantages

(Naphad, 2022) describes the following advantages of queues:"

- A large amount of data can be managed efficiently with ease.
- Operations such as insertion and deletion can be performed with ease as it follows the first in first out rule.
- Queues are useful when a particular service is used by multiple consumers.
- Queues are fast in speed for data inter-process communication.
- Queues can be used in the implementation of other data structures."

4.2 Disadvantages

- Naphad (2022) describes the following disadvantages of queues:
- The operations such as insertion and deletion of elements from the middle are time consuming.
- Limited Space.
- In a classical queue, a new element can only be inserted when the existing elements are deleted from the queue.
- Searching an element takes O(N) time.
- Maximum size of a queue must be defined prior.

Part 5

Applications

5.1 Application areas

- Queues are used in different areas as far as data structures are concerned. (*Applications of Queue Data Structure*, 2011) provides a summary of the areas queues are in use:”
- **Task Scheduling**: Queues can be used to schedule tasks based on priority or the order in which they were received.
- **Resource Allocation:** Queues can be used to manage and allocate resources, such as printers or CPU processing time.
- **Batch Processing**: Queues can be used to handle batch processing jobs, such as data analysis or image rendering.
- **Message Buffering**: Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.

5.1 Application areas

- **Event Handling**: Queues can be used to handle events in event-driven systems, such as GUI applications or simulation systems.
- **Traffic Management**: Queues can be used to manage traffic flow in transportation systems, such as airport control systems or road networks.
- **Operating systems:** Operating systems often use queues to manage processes and resources. For example, a process scheduler might use a queue to manage the order in which processes are executed.
- **Network protocols:** Network protocols like TCP and UDP use queues to manage packets that are transmitted over the network. Queues can help to ensure that packets are delivered in the correct order and at the appropriate rate.

5.1 Application areas

- **Printer queues :**In printing systems, queues are used to manage the order in which print jobs are processed. Jobs are added to the queue as they are submitted, and the printer processes them in the order they were received.
- **Web servers:** Web servers use queues to manage incoming requests from clients. Requests are added to the queue as they are received, and they are processed by the server in the order they were received.
- **Breadth-first search algorithm:** The breadth-first search algorithm uses a queue to explore nodes in a graph level-by-level. The algorithm starts at a given node, adds its neighbors to the queue, and then processes each neighbor in turn."

5.2 Specific applications

- (*Applications of Queue Data Structure*, 2011) also add the following specific applications of queues: "
- Applied as waiting lists for a single shared resource like CPU, Disk, and Printer.
- Applied as buffers on MP3 players and portable CD players.
- Applied on Operating system to handle the interruption.
- Applied to add a song at the end or to play from the front.
- Applied on WhatsApp when we send messages to our friends and they don't have an internet connection then these messages are queued on the server of WhatsApp.
- Traffic software (Each light gets on one by one after every time of interval of time.)"

Summary

- A queue is an ordered list in which insertions are done at one end (*rear*) and deletions are done at other end (*front*). The first element to be inserted is the first one to be deleted. Hence, it is called First in First out (FIFO) or Last in Last out (LILO) list."
- Main operations of a queue are enqueue (to add an element) and dequeue (to remove an element).
- Queues can be implemented in three data structures namely, Lists, Linked lists and arrays. Arrays can be further implemented as standard arrays or circular arrays.
- A priority queue is a queue in which each item is assigned a priority and items with a higher priority are removed before those with a lower priority, irrespective of when they were added.
- Priority queue can be implemented using the following data structures: arrays, Linked list, Heap data structure, and binary search tree.

References

- *Applications of Queue Data Structure*. (2011, March 1). GeeksforGeeks. https://www.geeksforgeeks.org/applications-of-queue-data-structure/
- Gupta, A. (2017, April 6). *Introduction to Circular Queue*. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-to-circular-queue/
- *Introduction and Array Implementation of Queue*. (2014, February 1). GeeksforGeeks. https://www.geeksforgeeks.org/introduction-and-array-implementation-of-queue/?ref=lbp
- *Linked List Implementation of Queue - javatpoint*. (2011). Www.javatpoint.com. https://www.javatpoint.com/linked-list-implementation-of-queue
- Lambert, K. (2019). *Fundamentals of Python: Data Structures*. Cengage Learning.

References

- Naphad, S. (2022, May 16). *Applications, Advantages and Disadvantages of Queue*. GeeksforGeeks. https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-queue/?ref=lbp
- Narasimha Karumanchi. (2017). *Data structures and algorithms made easy : to all my readers : concepts, problems, interview questions*. Careermonk Publications.
- Necaise, R. D. (2011). *Data structures and algorithms using Python*. John Wiley And Sons.
- Rajput. (2018, November 20). *Array implementation of queue (Simple) - GeeksforGeeks*. GeeksforGeeks. https://www.geeksforgeeks.org/array-implementation-of-queue-simple/