Queue

Chapter 7

Learning Objectives

Students will be able to:

- Describe queue concept, abstract data type and representation
- Implement queue with linked list, Python list and circular queue

Chapter Outline

1. Queue
   1) Queue Concept
   2) Queue Representation
   3) Queue Abstract Data Type (ADT)
2. Queue List
   1) Queue List Concept
   2) Queue list Application Programming Interface (API)
   3) Queue List Implementation
   4) Queue List Method's Performance
3. Queue Python List
   1) Queue Python List Concept
   2) Queue Python List Application Programming Interface (API)
   3) Queue Python List Implementation
   4) Queue Python List Method's Performance
4. Circular Queue
   1) Circular Queue Concept
   2) Circular Queue Application Programming Interface (API)
   3) Circular Queue Array Implementation
   4) Circular Queue Method's Performance

7

Queue

Section 1:
Queue

1) Queue Concept
2) Queue Representation
3) Queue Abstract Data Type (ADT)

Queue Concept

- It is a linear list in which data can only be inserted at one end, call the rear, and deleted from the other end, called the front
- These restrictions ensure that the data are processed through the queue in the order in which they are received.
- A queue is a First-In First-Out (FIFO) structure.
- Ex : A line of people waiting for the bus in a bus station, A list of waiting jobs to be processed by a computer.

Queue Concept

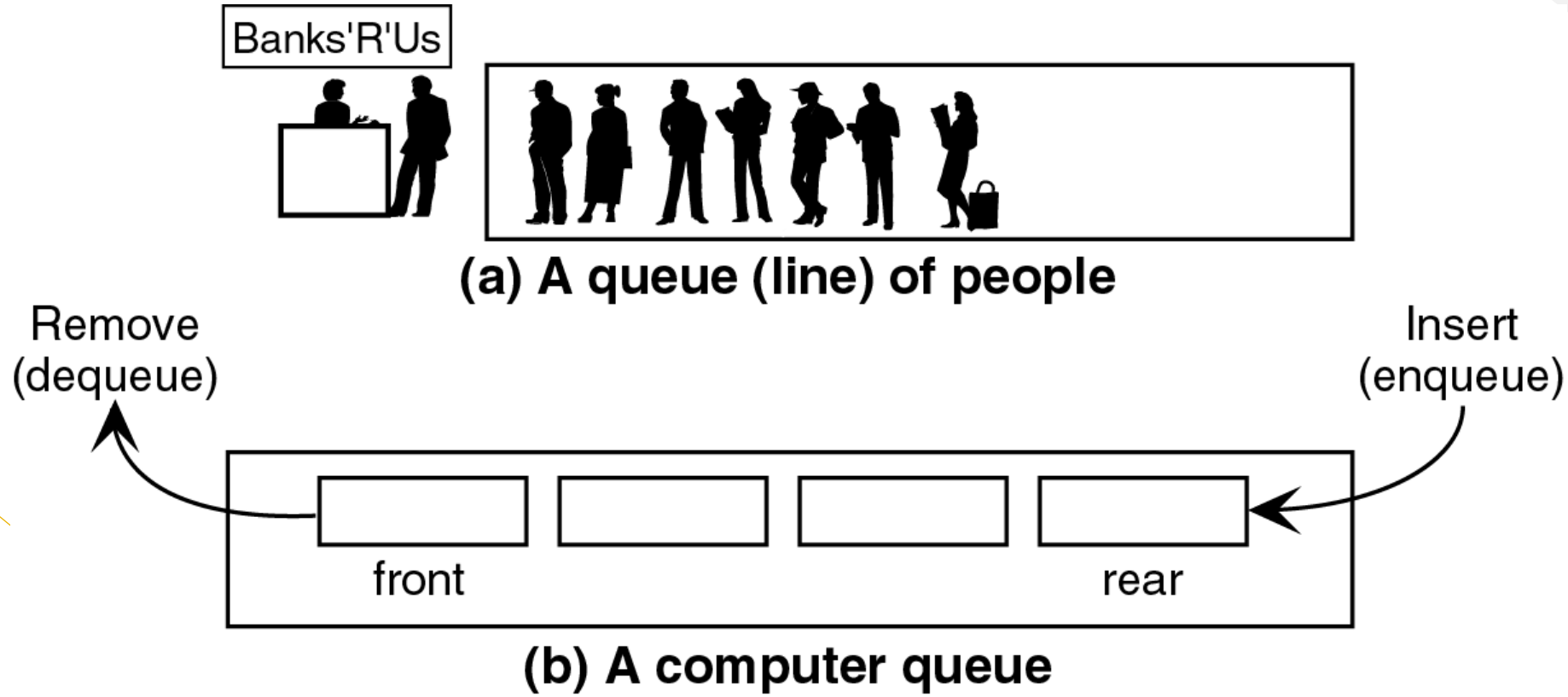
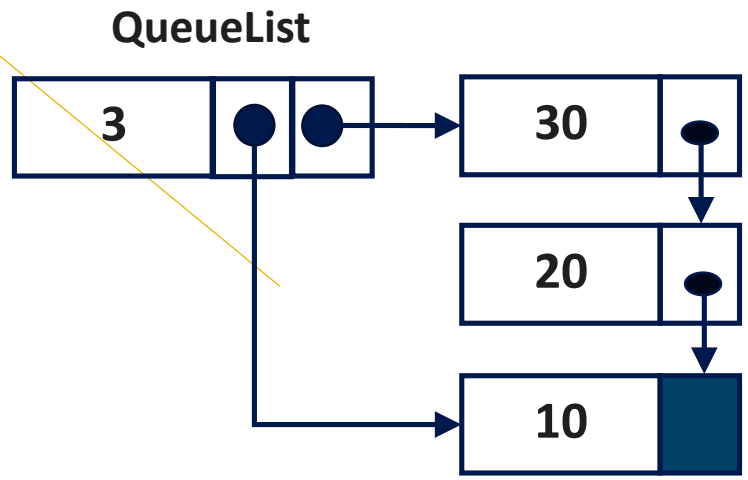


Fig 7-1 Queue concept [4]

Queue Representation

- It can be implemented with the primitive linear data structures:
  1. Linked list – Queue list
  2. Array Python list – Queue Python list
  3. Array – Circular queue array



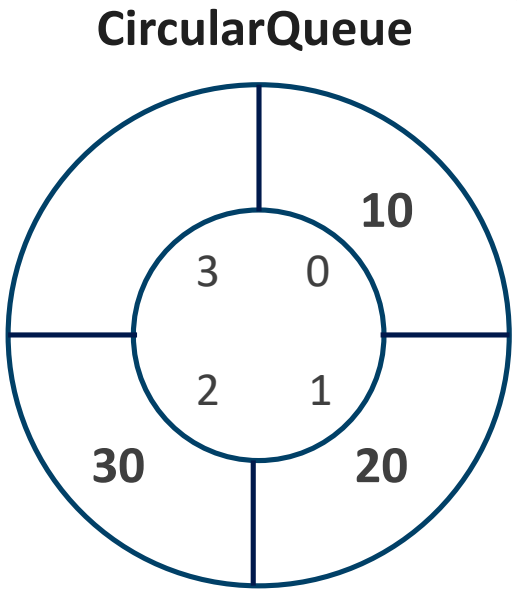


Fig 7-2 Queue representation

Queue ADT

- There are main operations - specially designed for queue:
  1. Enqueue: Insert operation which must be strictly inserted to the front of queue
  2. Dequeue: Remove the front item from the queue
  3. Front: Return the front item value or both value and its address
  4. Rear: Return the rear item value or both value and its address
  5. Empty queue: Return true if the queue is empty, otherwise is false

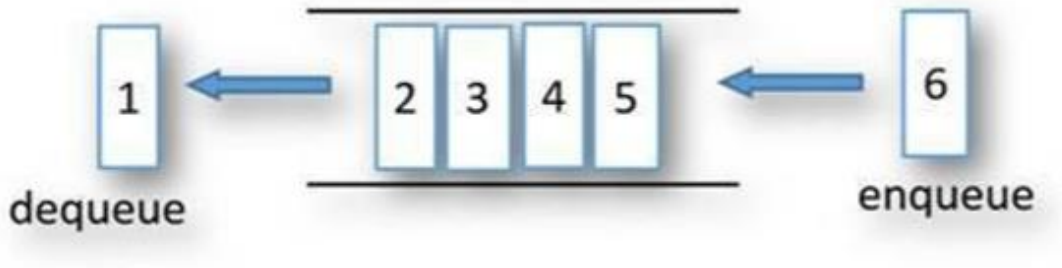


Fig 7-3 Queue Abstract Data Type [5]

Queue ADT

Enqueue operation

- A queue insertion which an item is added at the rear of queue
- Before adding, we must ensure that there is an available space to contain a new item otherwise it will be “Queue Overflow”.
- After enqueueing, the added item becomes the rear of queue.

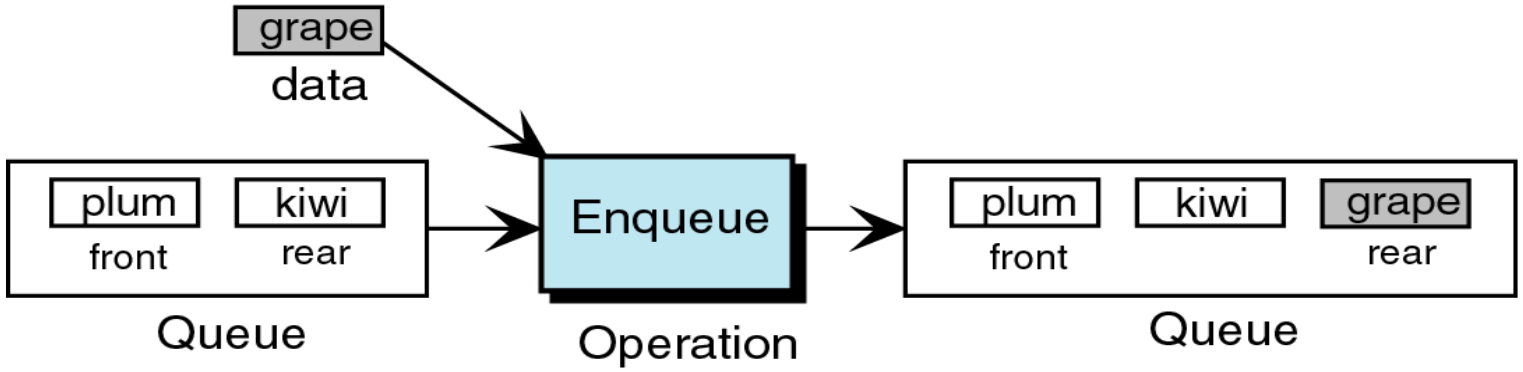


Fig 7-4 Enqueue operation [5]

Queue ADT

Dequeue operation

- A queue deletion will remove an item at the front of queue
- Before deleting, there is at least an item to be removed otherwise “queue Underflow” will be notified.
- After dequeening, the follower item becomes the front of queue.

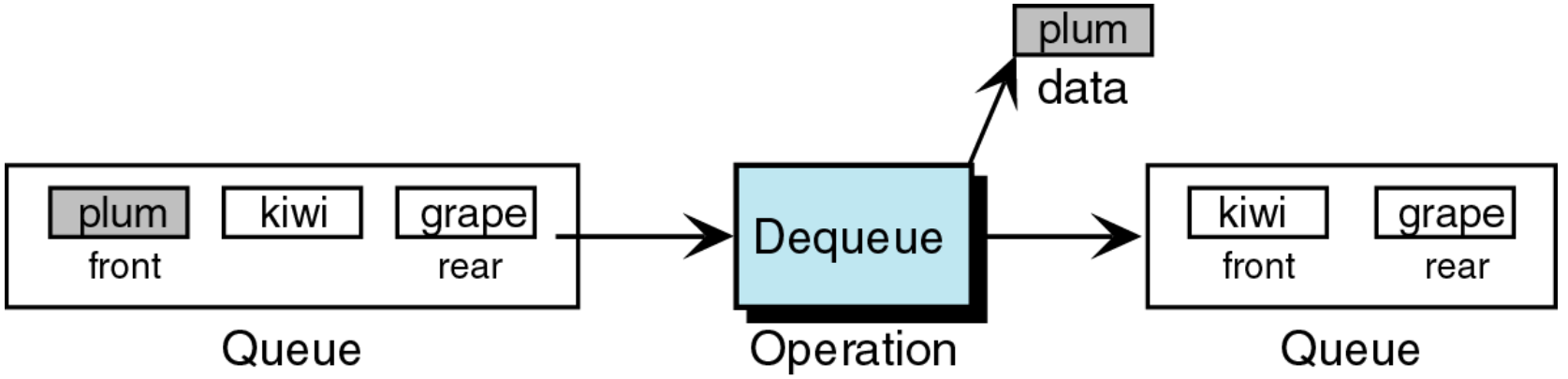


Fig 7-5 Dequeue operation [5]

Queue ADT

Front of queue

- Retrieving the item at the front and return – not delete, either
  1. Value or
  2. Pair of value and address.
- If the queue is empty, return a notification.

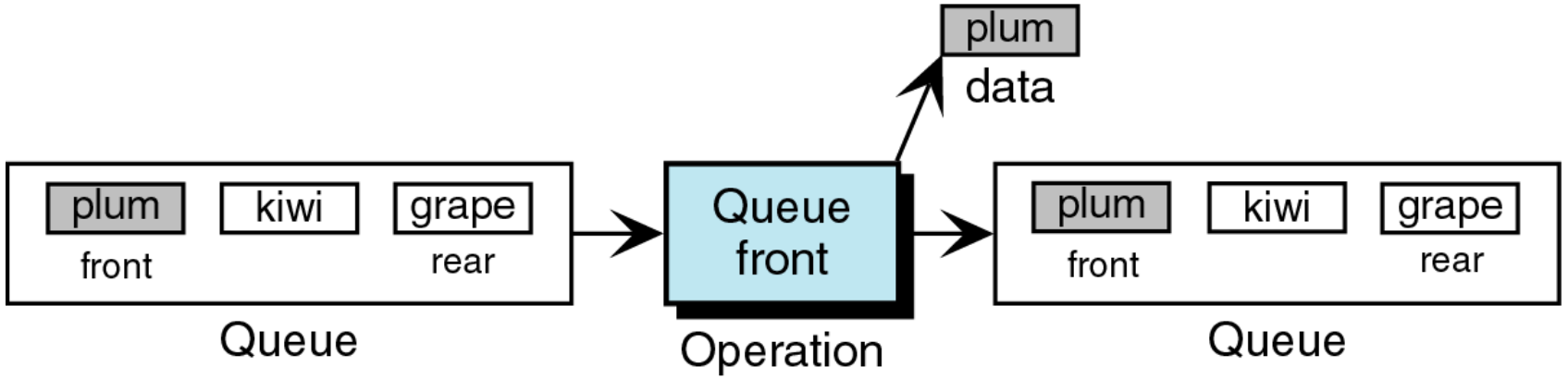


Fig 7-6 Front operation [5]

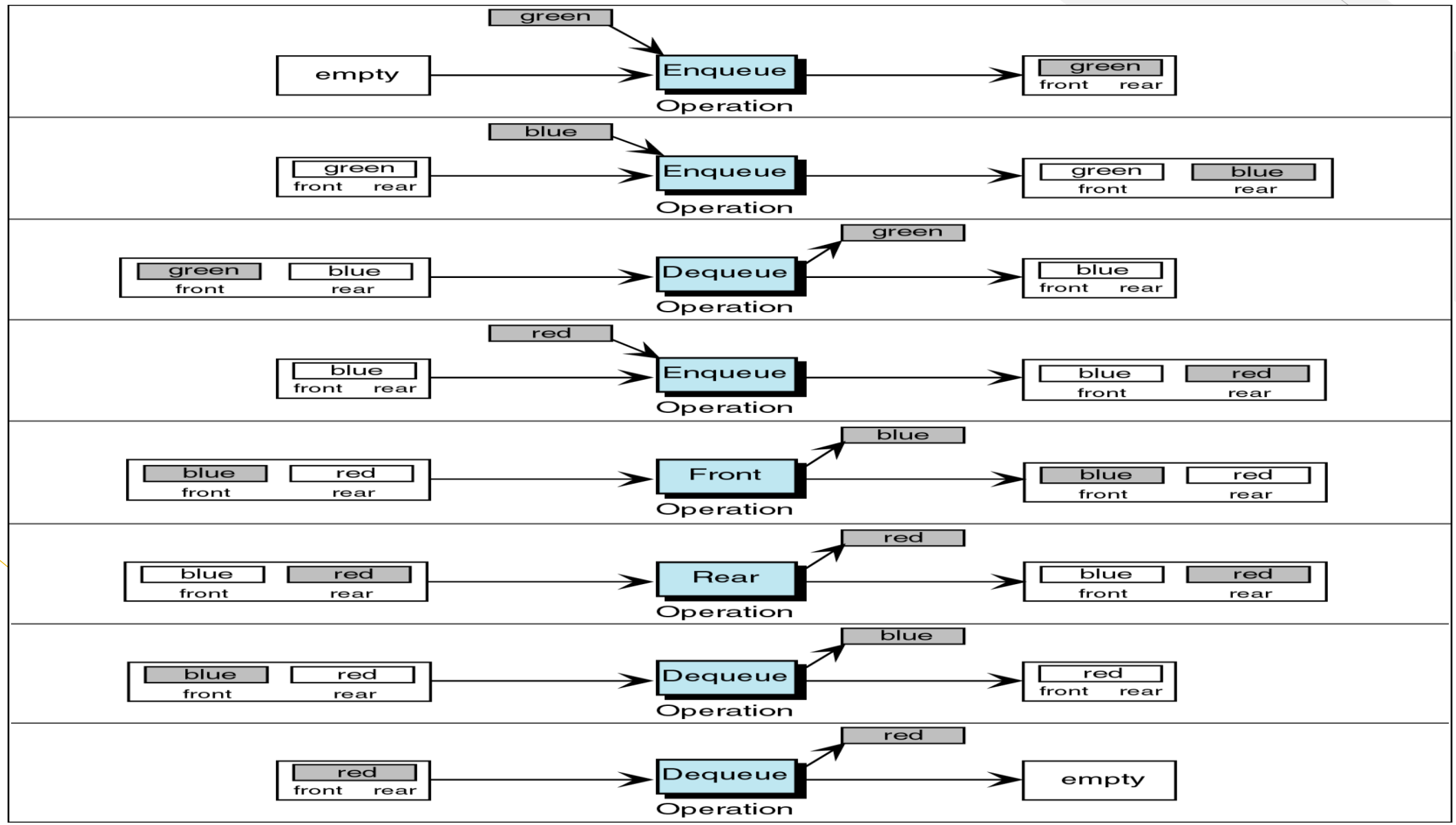


Fig 7-7 Example of queue operations [5]

7

Queue

Section 2:
Queue List

1) Queue List Concept
2) Queue list Application Programming Interface (API)
3) Queue List Implementation
4) Queue List Method’s Performance

Queue List Concept

- Queue is implemented with singly linked list whose head is front.
- Implementing queue list is chosen when the space utilization and maintenance cost – insert, update and delete is highly concerned.

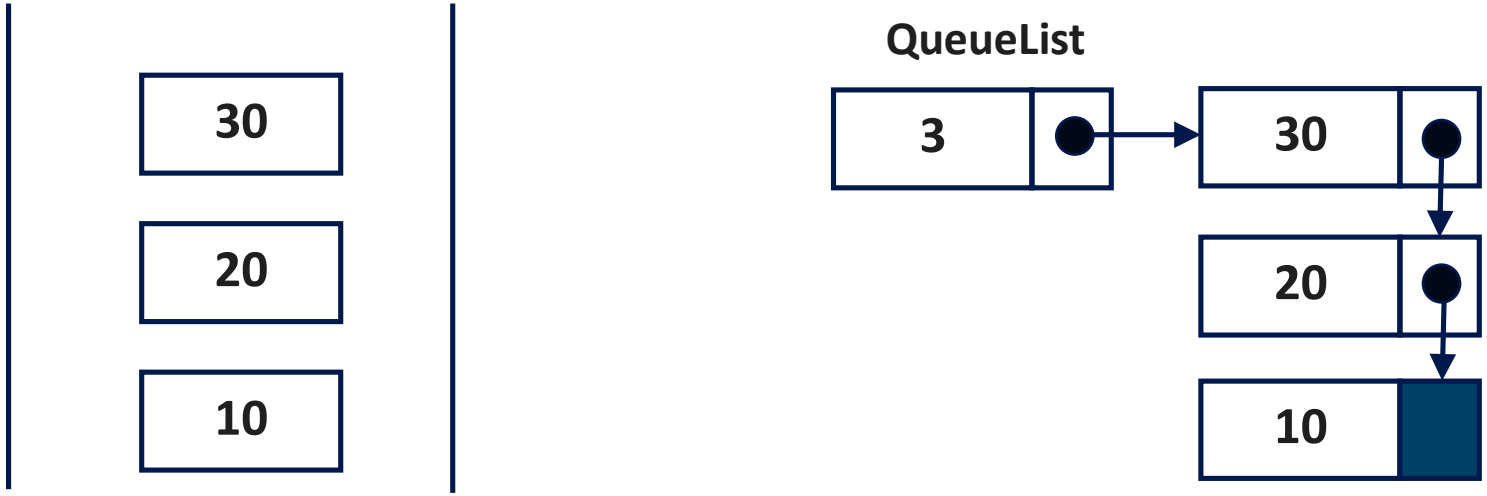


Fig 7-8 Conceptual and Physical view of queue list

Queue List Concept

Header node

- The head attribute is renamed as the Front to contain address of the Front item.
- Like linked list, header node can be designed to contain various useful information.

```
class QueueList:
  def __init__(self): #Queue List constructor
    self.front=None
    self.rear=None
    self.count=0
```

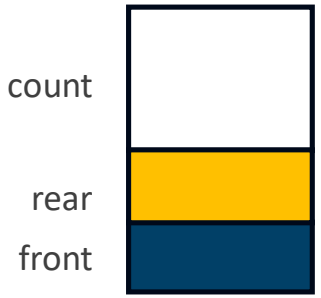


Fig 7-9 Header nod of queue list

Queue List API

- **Create a new queue list**: Initial an empty queue linked list
- **DisplayQueue**: Visit every data node in the queue list for serving an operation such as print, search, etc.
- **IsEmpty**: Check whether the existing queue list contains at least a node or not. If it is empty, return true. Otherwise, return false
- **Enqueue**: Insert a new node into the front of queue list
- **Dequeue**: Remove a target node from the queue Front
- **Front**: Return value at the front of queue list

Queue List API

Create a new queue list

- The header node contains address of the front data node and must be identified as an instance of queue list.

```
1  class Node:
2    def __init__(self, data, next): # node constructor
3      self.data = data
4      self.next = next
5
6  class QueueList:
7    def __init__(self): # Queue list constructor
8      self.front = None
9      self.rear = None
10     self.count = 0
```

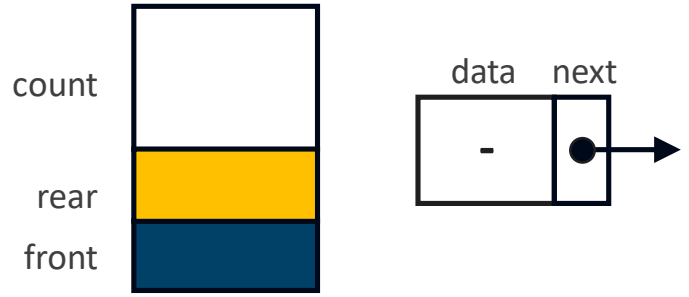


Fig 7-10 Queue list constructor

Queue List API

Traverse queue list

- It traverses entire nodes in the queue list with a variable to walk through the queue list until to the end.

```
1  def DisplayQueue(self): #Print entire data nodes
2      curNode = self.front
3      print("Print stack contains:", self.count, "nodes")
4      while curNode is not None:
5          print(curNode.data)
6          curNode = curNode.next
```

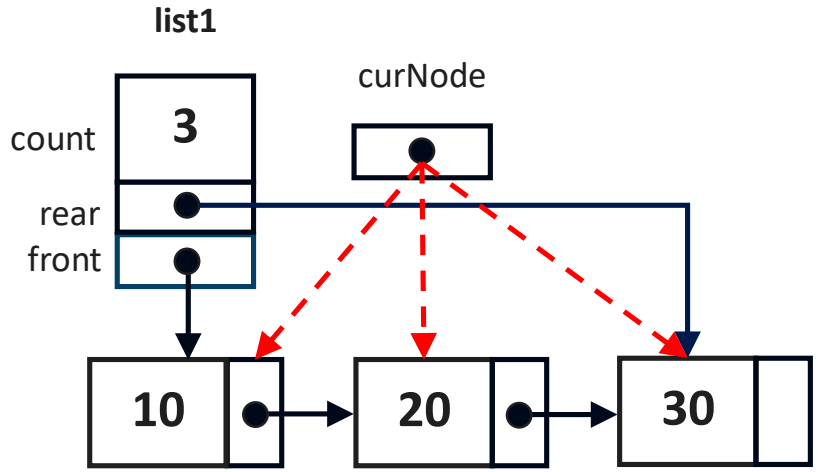


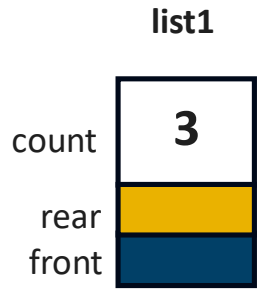
Fig 7-11 Traverse queue list

Queue List API

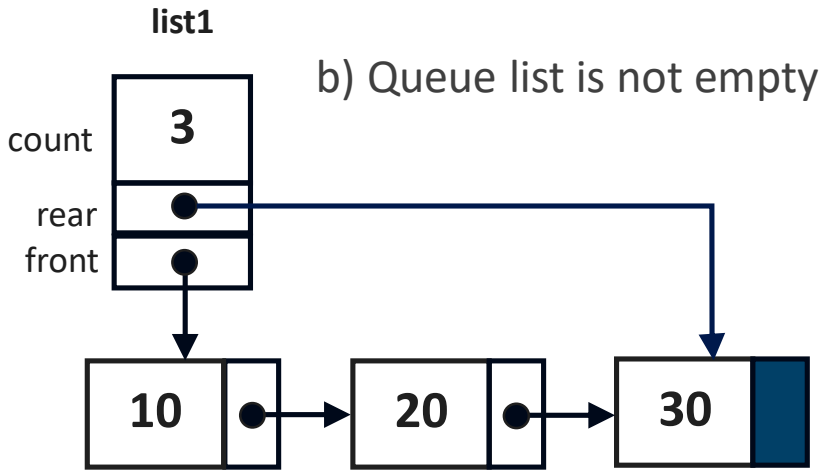
Empty queue

- Verify whether queue is empty or not

```
1  def IsEmpty(self):
2    if self.front is not None:
3      return False
4    else:
5      return True
```



a) Empty queue list



b) Queue list is not empty

Fig 7-12 Empty queue list

Queue List API

Queue list insertion

- Insert a new node to the queue list
- There is only a case can be used:
  1. ~~Insert at head~~
  2. Insert at end or tail: Enqueue
  3. ~~Insert any node~~

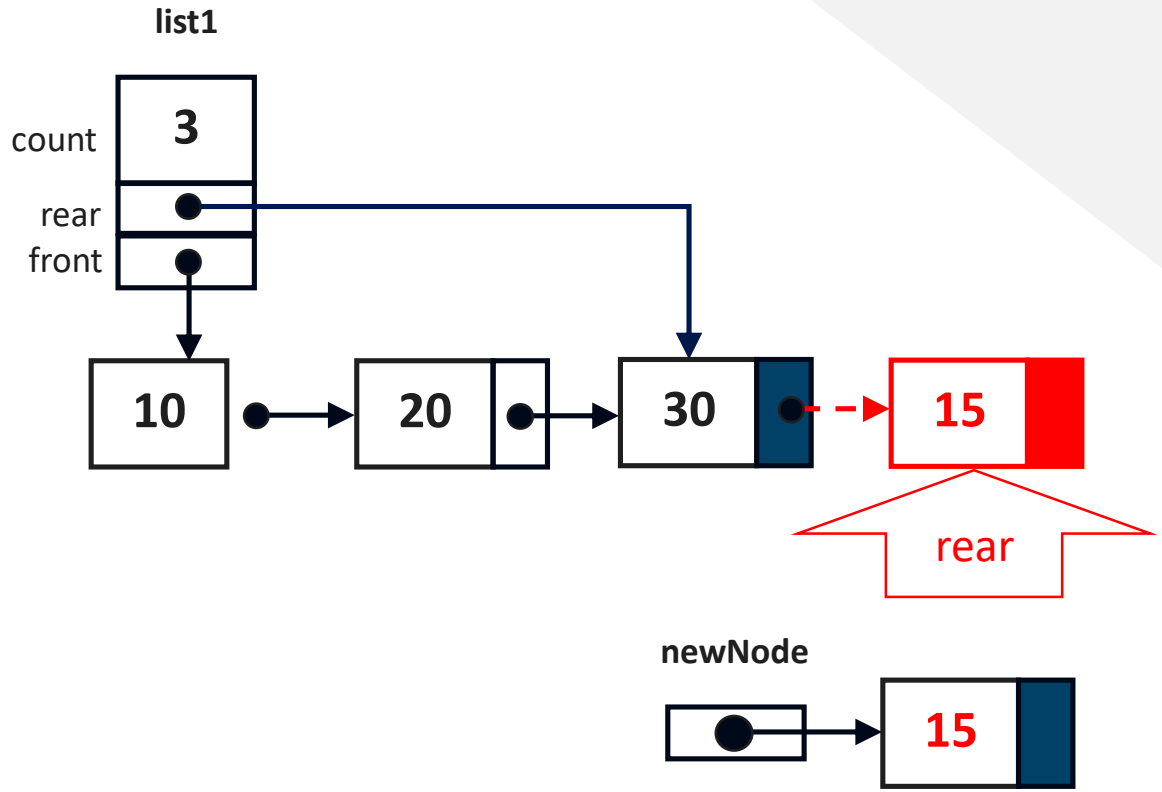


Fig 7-13 Queue list insertion

Queue List API

Enqueue

- Insert a node at the end of queue

```
1  def Enqueue(self, data): #Insert new node to end
2      newNode = Node(data, None)
3      #self.rear = self.front
4      if self.rear is None: #Empty list
5          self.rear = newNode
6          self.front = newNode
7      else: #Existing list which rear point to last node
8          self.rear.next = newNode
9          self.rear= self.rear.next
10     self.count += 1
```

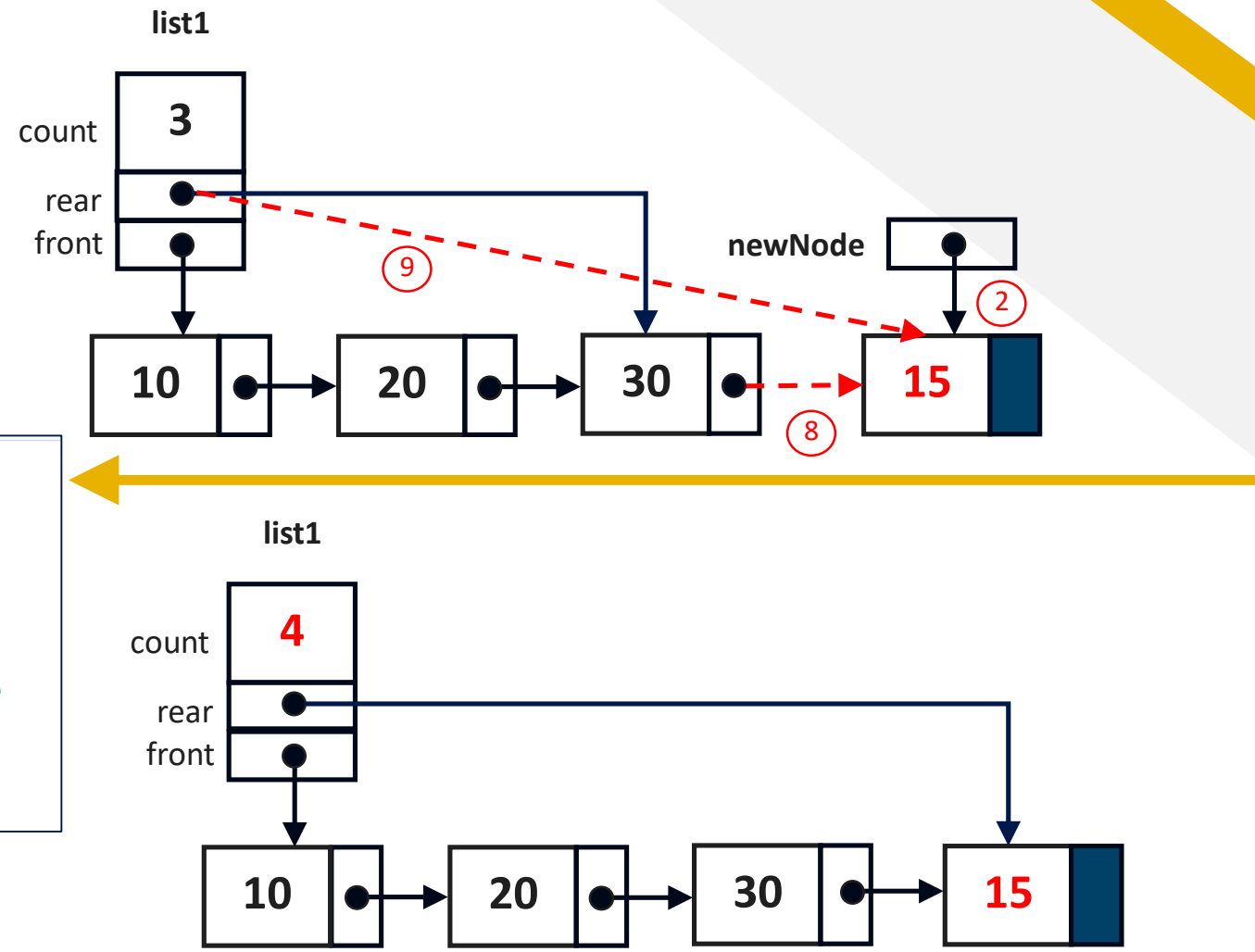


Fig 7-14 Queue list enqueue

Queue List API

Queue list deletion

There is only a case for the queue list:

1. Delete at head : Dequeue
2. ~~Delete at end or tail : Dequeue~~
3. ~~Delete any node~~

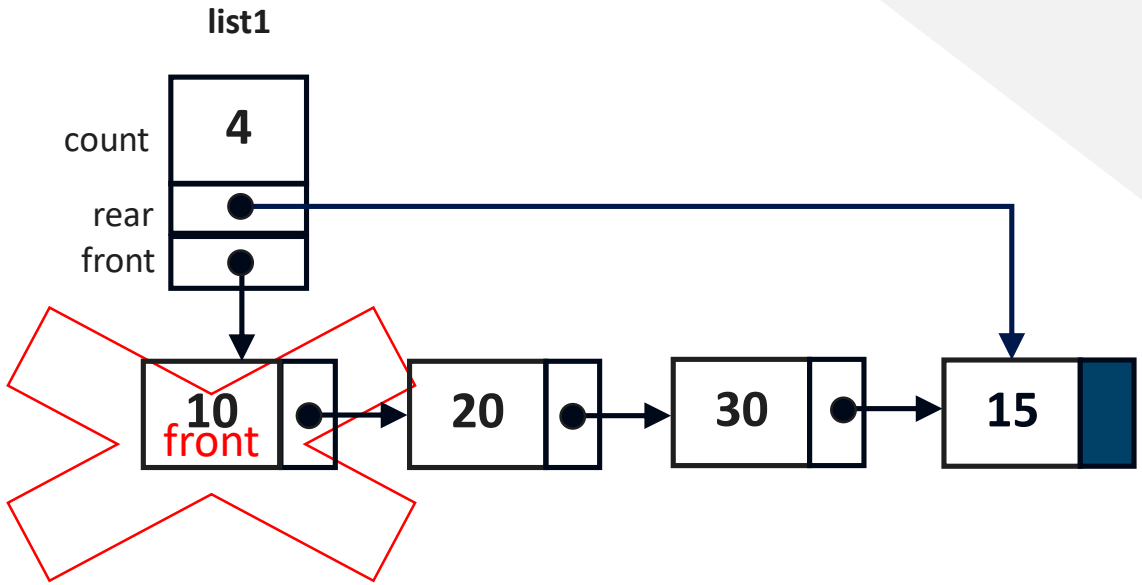


Fig 7-15 Queue list deletion

Queue List API

Dequeue

- Delete the front node

```
1  def Dequeue(self): #Delete and return a node at front
2    if self.IsEmpty()==False:
3      dequeueValue = self.front.data
4      self.front = self.front.next
5      self.count -= 1
6    else:
7      print("The queue is empty")
8    return dequeueValue
```

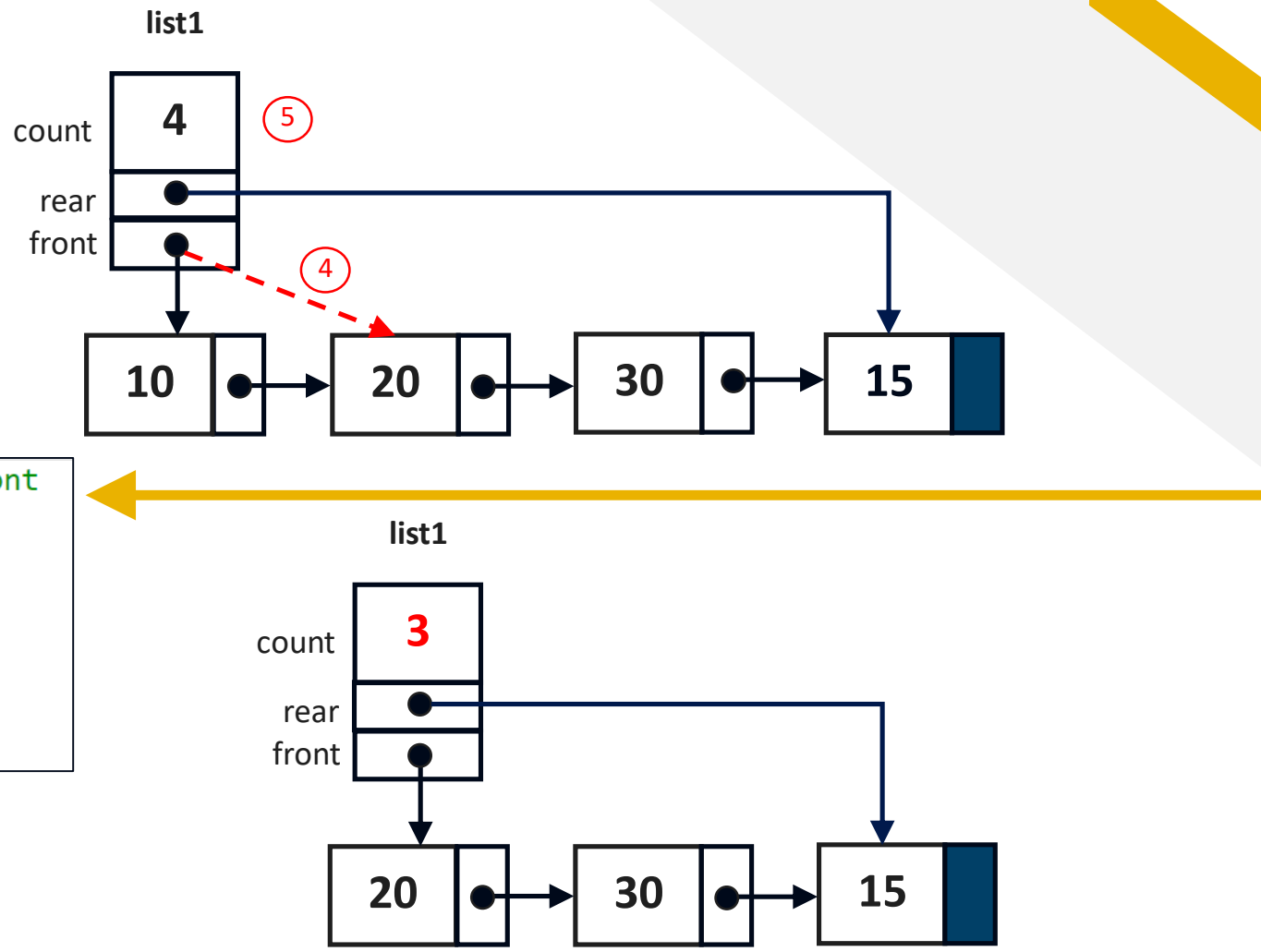


Fig 7-16 Queue list dequeue

Queue List API

Front

- Retrieve and return the queue front –
  1. Node value
  2. Node address

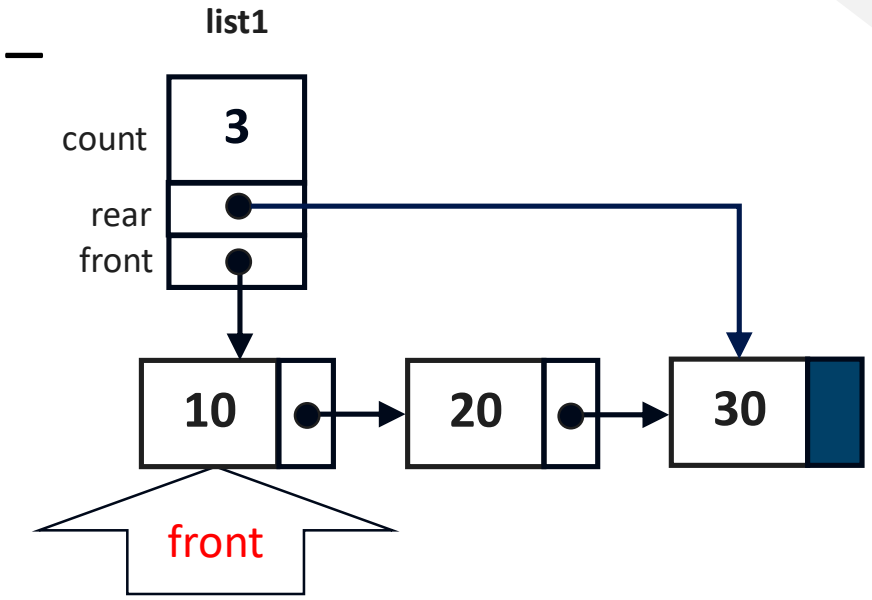


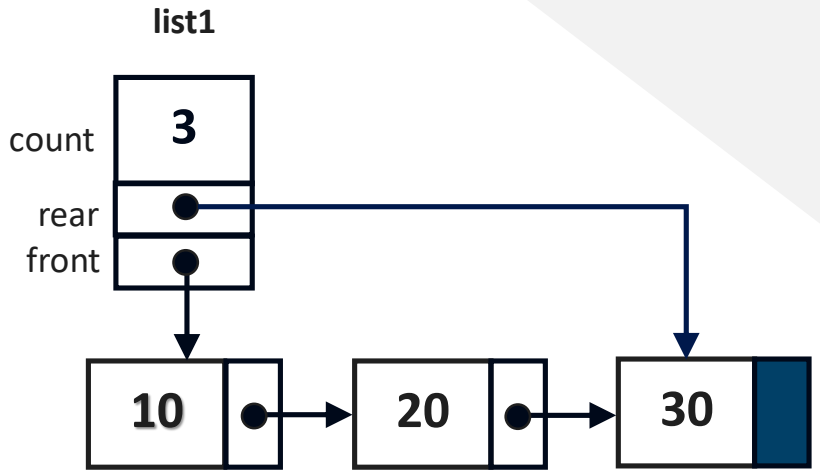
Fig 7-17 Retrieve queue list front

Queue List API

Front

- Retrieve and return the front node value

```
1  def Front(self): #Retrieve and return a node at front
2    if self.IsEmpty()==False:
3      return self.front.data
4    else:
5      return "The front of queue list is none"
```



Front node = 10

Fig 7-18 Queue list front

Queue List API

Rear

- Retrieve and return the queue rear –
  1. Node value
  2. Node address

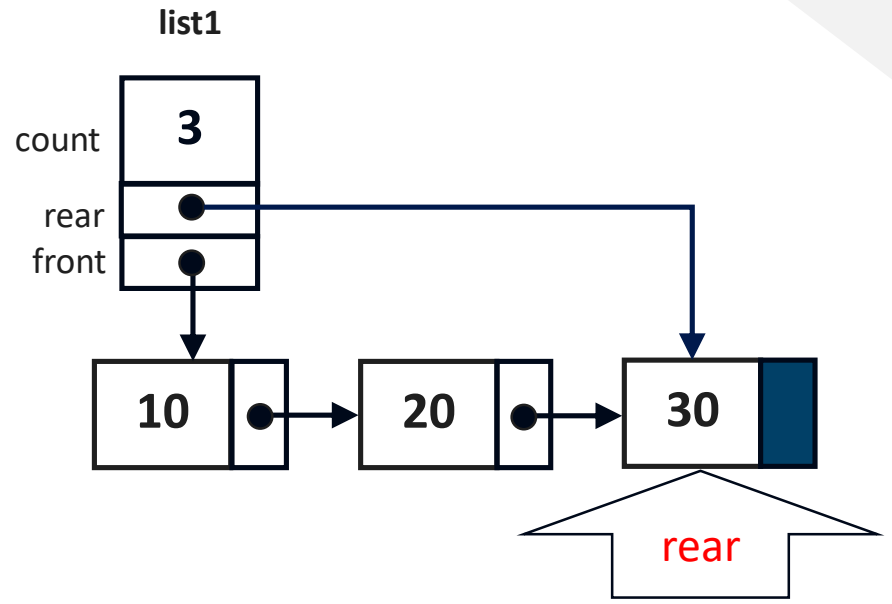


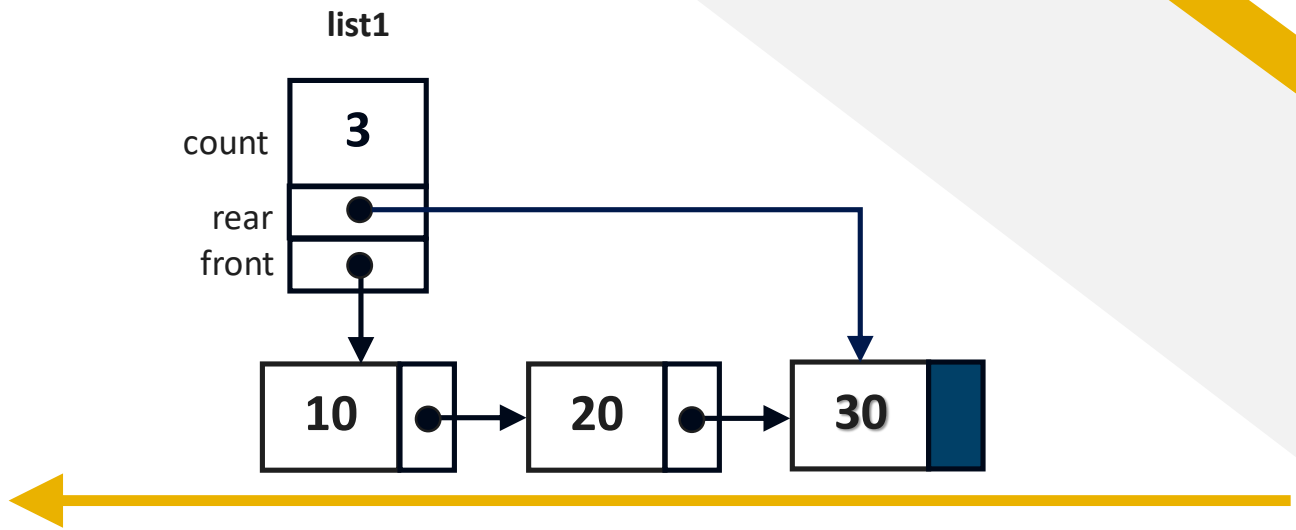
Fig 7-19 Retrieve queue list rear

Queue List API

Rear

- Retrieve and return the front node value

```
1  def Rear(self): #Retrieve and return a node at rear
2    if self.IsEmpty()==False:
3      return self.rear.data
4    else:
5      return "The rear of queue list is none"
```



Rear node = 30

Fig 7-20 Queue list rear

```
1  class Node:
2    def __init__(self, data, next): # node constructor
3      self.data = data
4      self.next = next
5
6  class QueueList:
7    def __init__(self): # Queue list constructor
8      self.front = None
9      self.rear = None
10     self.count = 0
11
12   def IsEmpty(self):
13     if self.front is not None:
14       return False
15     else:
16       return True
17
18   def DisplayQueue(self): #Print entire data nodes
19     curNode = self.front
20     print("Print stack contains:", self.count, "nodes")
21     while curNode is not None:
22       print(curNode.data)
23       curNode = curNode.next
24
```

Tutorial 1: Queue List Implementation (Cont.)

2.3

```
25  def Enqueue(self, data): #Insert new node to end
26    newNode = Node(data, None)
27    #self.rear = self.front
28    if self.rear is None: #Empty list
29      self.rear = newNode
30      self.front = newNode
31    else: #Existing list which rear point to last node
32      self.rear.next = newNode
33      self.rear= self.rear.next
34    self.count += 1
35
36  def Dequeue(self): #Delete and return a node at front
37    if self.IsEmpty()==False:
38      DeQueueValue = self.front.data
39      self.front = self.front.next
40      self.count -= 1
41    else:
42      print("The queue is empty")
43    return DeQueueValue
44
```

```
45  def Front(self): #Retrieve and return a node at front
46    if self.IsEmpty()==False:
47      return self.front.data
48    else:
49      return "The front of queue list is none"
50
51  def Rear(self): #Retrieve and return a node at rear
52    if self.IsEmpty()==False:
53      return self.rear.data
54    else:
55      return "The rear of queue list is none"
56
57  def listSize(self): #Return node counter
58    return self.count
```

Tutorial 1: Queue List Implementation (Cont.)

2.3

```
59 #---
60 list1 = QueueList() #Create a new stack list
61 #Is Queue list empty?
62 if list1.IsEmpty():
63   print("Queue list is empty")
64 else:
65   print("Queue list is not empty")
66 #Push a new node
67 list1.Enqueue(10)
68 list1.Enqueue(20)
69 list1.Enqueue(30)
70 list1.Enqueue(15)
71 list1.DisplayQueue() #Display the list
72 #Dequeue and return the front node
73 print("The node - ", list1.Dequeue(), ", is Dequeue.")
74 list1.DisplayQueue() #Display the list
75 #Return a front node
76 print("The front of queue list is : ", list1.Front())
77 #Return a rear node
78 print("The rear of queue list is : ", list1.Rear())
```

```
Queue list is empty
Print stack contains: 4 nodes
10
20
30
15
The node -  10 , is Dequeue.
Print stack contains: 3 nodes
20
30
15
The front of queue list is :  20
The rear of queue list is :  15
```

Queue List Methods' Performance

Analysis methods' peroformance of queue list

Methods / Operations	Description	Big-O
Create queue list	Initial default value to queue list class – including front and count.	O(1)
Traverse queue list	Moving to the end of queue list which depend on total nodes in the queue list.	O(n)
Empty queue list	It is only check value of the queue front whether is null or not.	O(1)
Enqueue	There are only update linkage of the front and new node without any traversal.	O(1)
Dequeue	The update linkage of the front to a successor node occurs without any shift or slide cost	O(1)
Front	Retrieve and return the front node	O(1)
Rear	Retrieve and return the rear node	O(1)

Table 7-1 Analysis queue list operations

7

Queue

Section 3:
Queue Python List

1) Queue Python List Concept
2) Queue Python List Application Programming Interface (API)
3) Queue Python List Implementation
4) Queue Python List Method's Performance

Queue Python List Concept

- Python list provides set of methods that can easily implement the queue structure.
- We usually design head of array as front and tail of array as read.

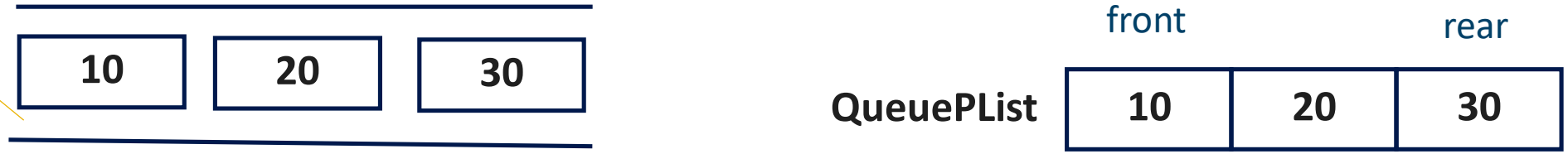


Fig 7-21 Conceptual and Physical view of queue array

Queue Python List Concept

- Python list handles and saves the shift or slide item cost usually occurs in traditional array structure.
- All deletion provided methods such as del operator, pop()

```
x = Q.dequeue()
Q.enqueue( 21 )
Q.enqueue( 74 )
```

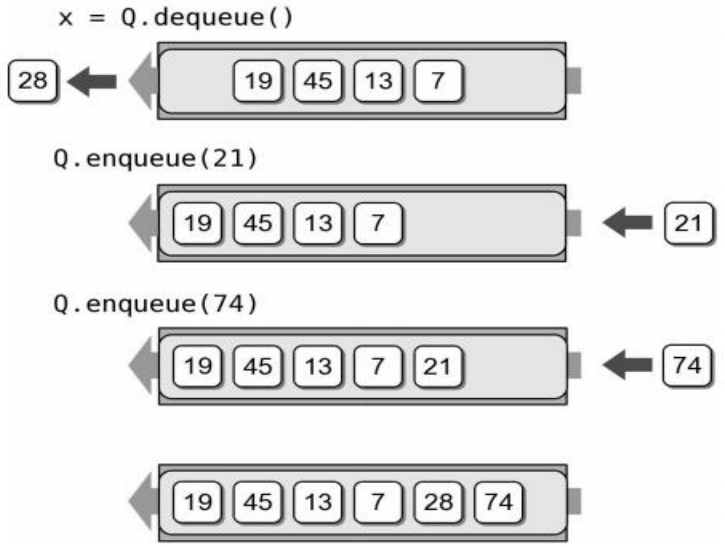


Fig 7-22 Example of queue Python list operations [3]

Queue Python List API

- **Create a new queue**: Initial an new queue Python list
- **Traverse queue**: Visit every data items in the queue Python list for serving an operation such as print, search, etc.
- **Empty queue** : Check whether the existing queue Python list contains at least a node or not. If it is empty, return true. Otherwise, return false.
- **Enqueue**: Insert a new item into the front of queue Python list
- **Dequeue**: Remove an item from the queue front
- **Front**: Retrieve and return item at the front of queue Python list
- **Rear**: Retrieve and return item at the rear of queue Python list

Queue Python List API

Create a new queue Python list

- Create a new Python list as a queue in the constructor

```
1 class QueuePythonList:
2   def __init__(self): # Queue Python list constructor
3     self.queuePList = []
```

Fig 7-23 Queue Python list constructor

Queue Python List API

Traverse queue Python list

- It traverses entire items in the queue Python list.
- Walk through the queue Python list until reach to the end of list

```
1  def DisplayQueue(self): #Print entire data items
2    print("Queue contains:", self.QueueSize(), "items")
3    i=0
4    while i<=len(self.queuePList)-1:
5      print(self.queuePList[i])
6      i+=1
```

Fig 7-24 Traverse queue Python list

Queue Python List API

Empty queue Python list

- Check status of queue whether is empty or not

```
1   def IsEmpty(self):
2       return self.queuePList == []
```

a) Empty queue list

b) Queue is not empty

Fig 7-25 Queue Python list Empty

Queue Python List API

Enqueue

- Call method append ( ) of Python list to insert a new item to the list

```
1  def Enqueue(self, data): #Insert a new item to list rear
2    self.queuePList.append(data)
```

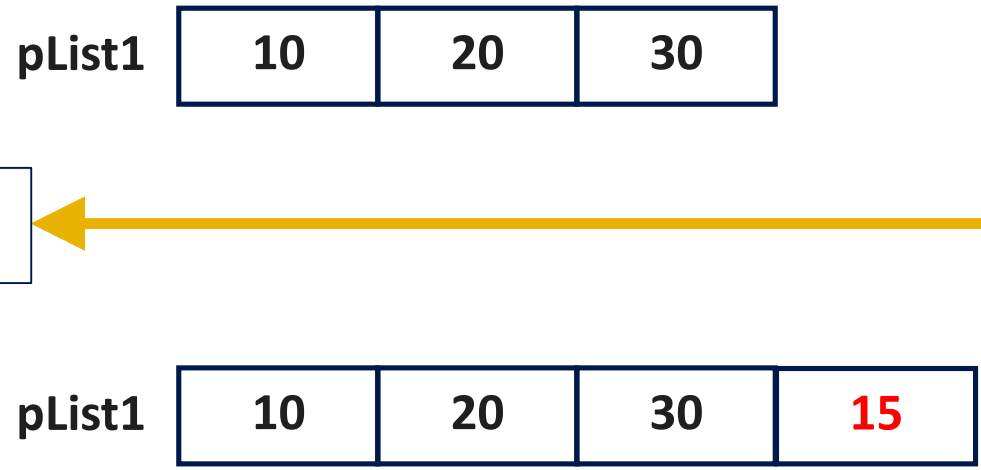


Fig 7-26 Queue Python list enqueue

Queue Python List API

Dequeue

- Call method pop () of Python list to remove the front item which is at index 0 because Python list shifts/slides all remaining items to the left and free space at the end after removing an item.

```
1  def Dequeue(self): #Delete a front data
2    if self.IsEmpty()==False:
3      #Front is always at index 0
4      dequeueValue = self.queuePList[0]
5      self.queuePList.pop(0)
6    else: #If queue is empty
7      print("Queue is underflow")
8    return dequeueValue
```

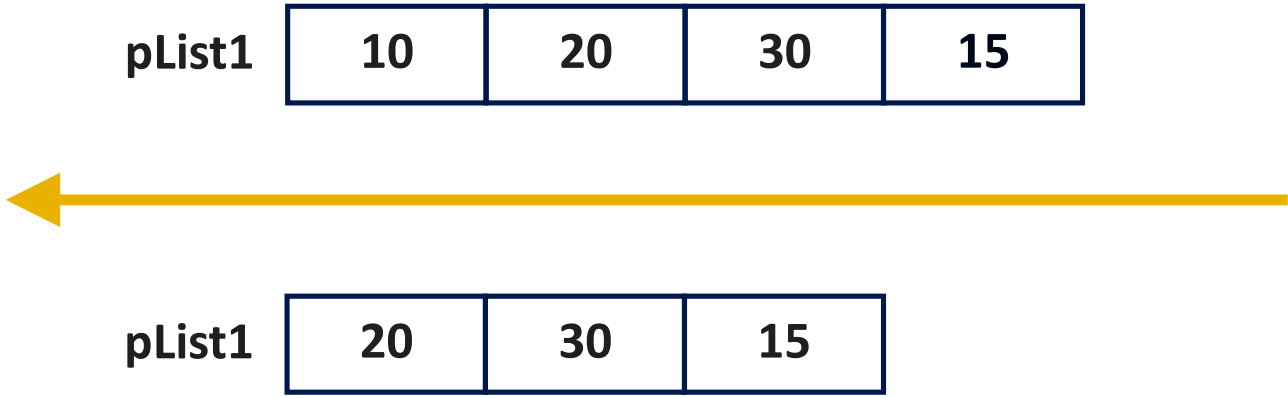


Fig 7-27 Queue Python list dequeue

Queue Python List API

Front

- All deletion methods provided by Python list manages item shifting make therefore front of queue Python list is always fixed at index 0.

```
1  def Font(self):
2    #Font of Phython list is always at index 0
3    return self.queuePList[0]
```

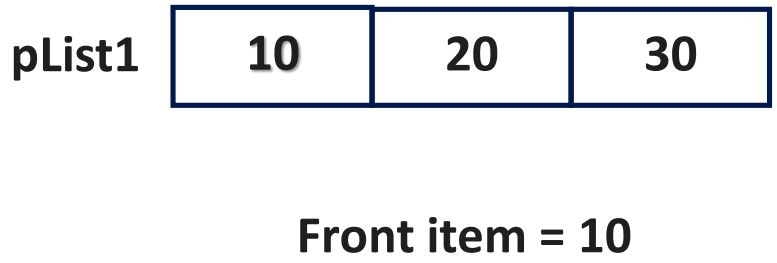


Fig 7-28 Queue Python list front

Queue Python List API

Rear

- All deletion methods provided by Python list manages item shifting make therefore front of queue Python list is always fixed at index 0.

```
1  def Rear(self):
2    return self.queuePList[len(self.queuePList)-1]
```

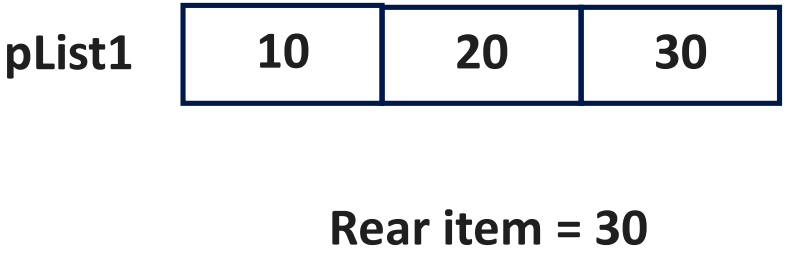


Fig 7-29 Queue Python list rear

```
1  class QueuePythonList:
2    def __init__(self): # Queue Python list constructor
3      self.queuePList = []
4
5    def IsEmpty(self):
6        return self.queuePList == []
7
8    def DisplayQueue(self): #Print entire data items
9      print("Queue contains:", self.QueueSize(), "items")
10     i=0
11     while i<=len(self.queuePList)-1:
12       print(self.queuePList[i])
13       i+=1
14
```

Tutorial 2: Queue Python List Implementation (Cont.)

```
15  def Enqueue(self, data): #Insert a new item to list rear
16    self.queuePList.append(data)
17
18  def Dequeue(self): #Delete a front data
19    if self.IsEmpty()==False:
20      dequeueValue = self.queuePList[0]
21      self.queuePList.pop(0)
22    else:
23      print("Queue is underflow")
24    return dequeueValue
25
26  def Font(self):
27    return self.queuePList[0]
28
29  def Rear(self):
30    return self.queuePList[len(self.queuePList)-1]
31
32  def QueueSize(self): #Return node counter
33    return len(self.queuePList)
```

```
#---
pList1 = QueuePythonList() #Create a new stack list
if pList1.IsEmpty(): #Is stack list empty?
  print("Queue array is empty")
else:
  print("Queue array is not empty")
#Push a new node
pList1.Enqueue(10)
pList1.Enqueue(20)
pList1.Enqueue(30)
pList1.Enqueue(15)
#array1.DisplayQueue() #Display the list
pList1.DisplayQueue() #Display the array queue
#DeQueue and return a front node
print ("The dequeue item is : ",pList1.Dequeue())
pList1.DisplayQueue() #Display the array queue
print ("The dequeue item is : ",pList1.Dequeue())
pList1.DisplayQueue() #Display the array queue
#Retrun the front element
print ("Font of queue value is : ",pList1.Font())
#Retrun the rear element
print ("Rear of queue value is : ",pList1.Rear())
```

```
Queue array is empty
Queue contains: 4 items
10
20
30
15
The dequeue item is :  10
Queue contains: 3 items
20
30
15
The dequeue item is :  20
Queue contains: 2 items
30
15
Font of queue value is :  30
Rear of queue value is :  15
```

Queue Python List Methods' Performance

Analysis methods' peroformance of queue Python list

Methods / Operations	Description	Big-O
Create a python list	Create a Python list as a new queue	O(1)
Traverse queue array	Update the array index to traverse every items of Python list	O(n)
Empty queue list	It is only check whether queue Python list is empty or not	O(1)
Enqueue	Append a new element to the end of queue Python list	O(1)
Dequeue	Remove the front item which is index 0 with item shift/slide supported by the Python list method.	O(n)
Front	Retrieve and return the front item at index 0	O(1)
Rear	Retrieve and return the rear item of Python list	O(1)

Table 7-2 Analysis of queue Python list operations

7

Queue

Section 4:
Circular Queue

1) Circular Queue Concept
2) Circular Queue Application Programming Interface (API)
3) Circular Queue Array Implementation
4) Circular Queue Method's Performance

Circular Queue Concept

- Python list provides an ordered collection mechanism and set of methods that can be easily implement the queue array.
- We usually design head of array as front and tail of array as read.

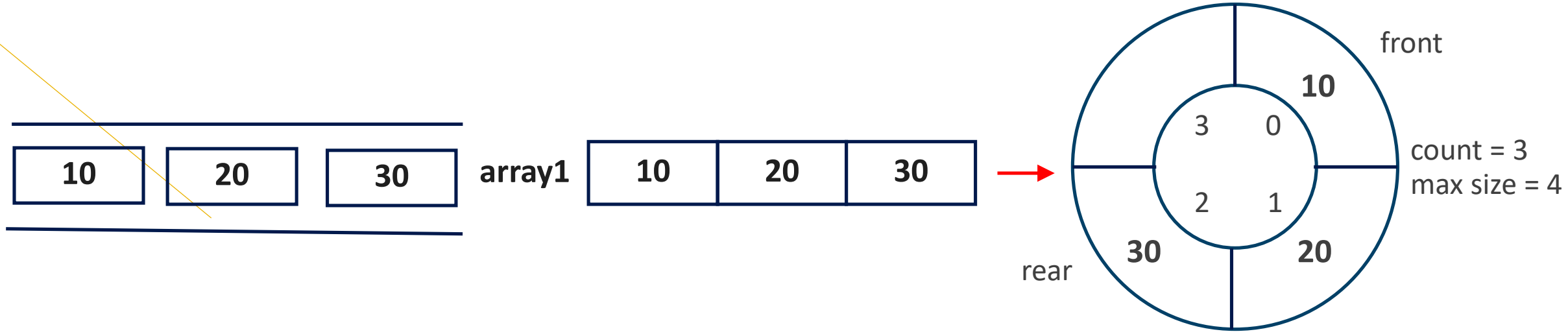


Fig 7-30 Conceptual and Physical view of queue array

Circular Queue Concept

- It is designed to get rid of the left spaces caused by Dequeue method implemented in a fixed size array which neglects the shift or slide operation.

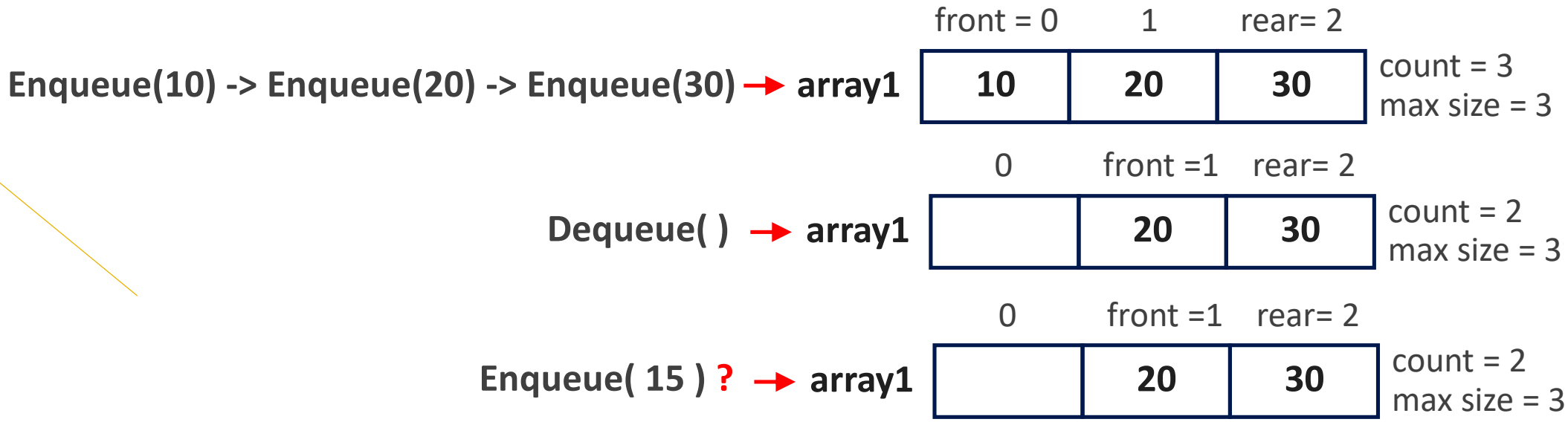


Fig 7-31 Conceptual and Physical view of circular queue

Circular Queue Concept

- Instead of shifting of slide elements, the circular queue wraps around index to the beginning to reuse Dequeue elements.

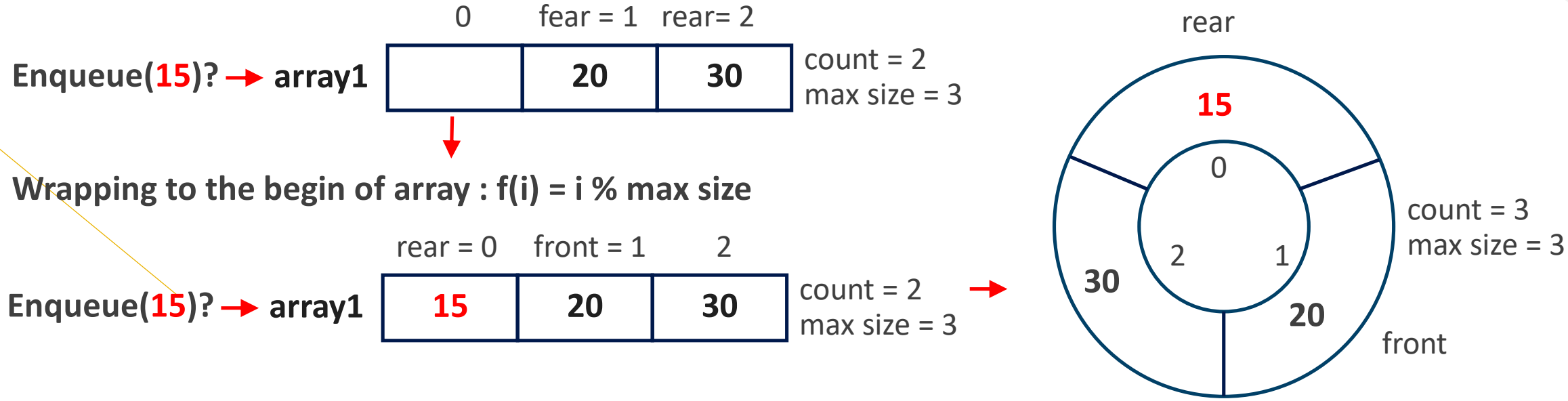


Fig 7-32 Circular queue concept

Circular Queue Concept

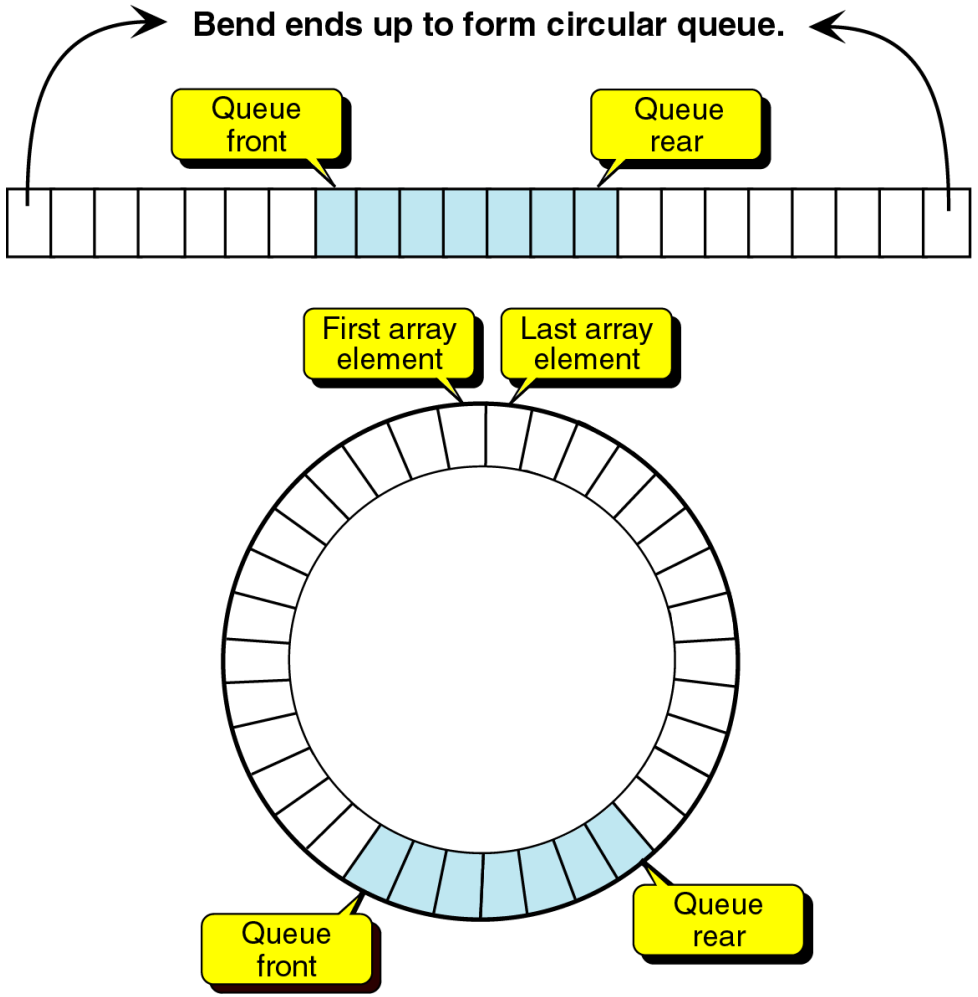


Fig 7-33 Wrapping circular queue

Circular Queue API

- **Create a new queue array**: Initial an empty queue
- **Traverse queue array**: Visit every data node in the queue
- **Empty queue array**: Check whether the queue is empty or not
- **Full queue array**: Validate a space of circular queue
- **Enqueue**: Insert a new element into the front of circular queue
- **Dequeue**: Remove the rear element from the circular queue
- **Front**: Retrieve and return the front element
- **Rear**: Retrieve and return the rear element

Circular Queue API

Create a new circular queue

- Create a Python list to represent a circular queue array
- Fix array size for bending it up to the front of circular queue
- Since Python handle shift or slide tasks for every insert and deletion operation, they must be disable in the circular queue.
- Initial a default value which will not be a data member is required.
- Since we must ignore insertion and deletion provided by Python, enqueue and dequeue are viewed as the element replacement, instead.

Circular Queue API

Create a new circular queue

- Initial default values when a Python list is initiated.
- Passing the max size of array or fixing is necessary for wrapping.
- The counter is needed to verify the full status of a circular queue.
- The front and rear will keep the index of front and rear elements.

Circular Queue API

Create a new circular queue

```
1  class QueueArray:
2    def __init__(self,maxSize): # Circular queue constructor
3      self.queueArr = []
4      self.front=0
5      self.rear=-1
6      self.count=0
7      self.maxSize= maxSize #Limit queue size
8      i=0
9      print("Circular queue - size :",self.maxSize)
10     while i<=maxSize-1:
11       #self.queueArr[i] = "None"
12       self.queueArr.insert(i, "None")
13       print(i, ".", self.queueArr[i])
14       i+=1
```

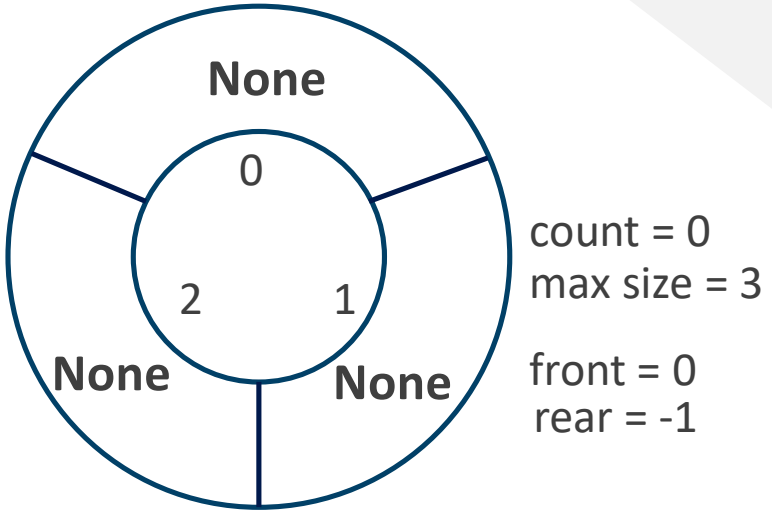


Fig 7-34 Circular queue constructor

Circular Queue API

Traverse circular queue

- It traverses entire elements of the circular queue.
- Like Python list, updating the array index is walking to the end of queue.

```
1  def DisplayQueue(self): #Print entire queue elements
2    print("Circular queue contains:", self.QueueSize(), "elements")
3    i=0
4    while i<=len(self.queueArr)-1:
5      print(i, ".", self.queueArr[i])
6      i+=1
```

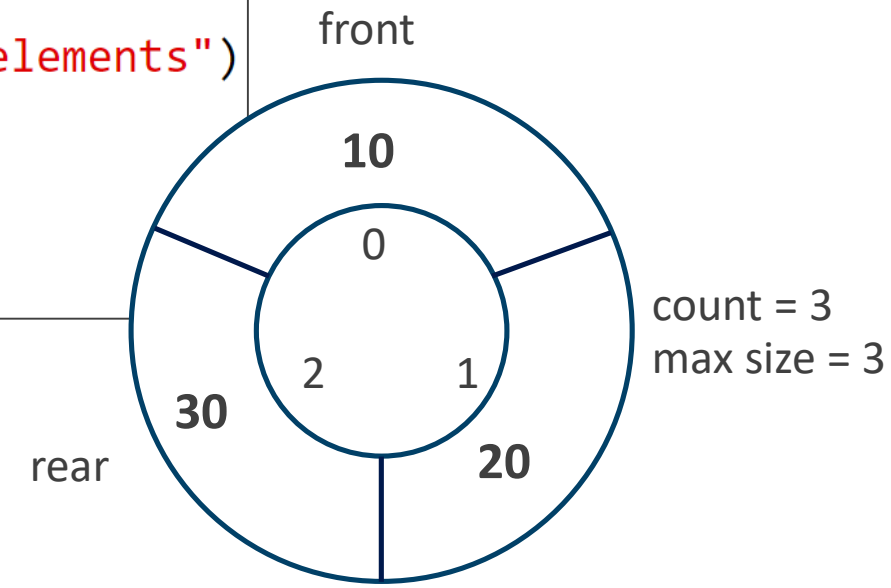


Fig 7-35 Traverse circular queue

Circular Queue API

Full circular queue

- Since it is represented as a fixed array, verify the circular queue whether it is full or not is necessary.

```
1  def IsFull(self): #Limit queue size, check size is required
2    return self.count == self.maxSize
```

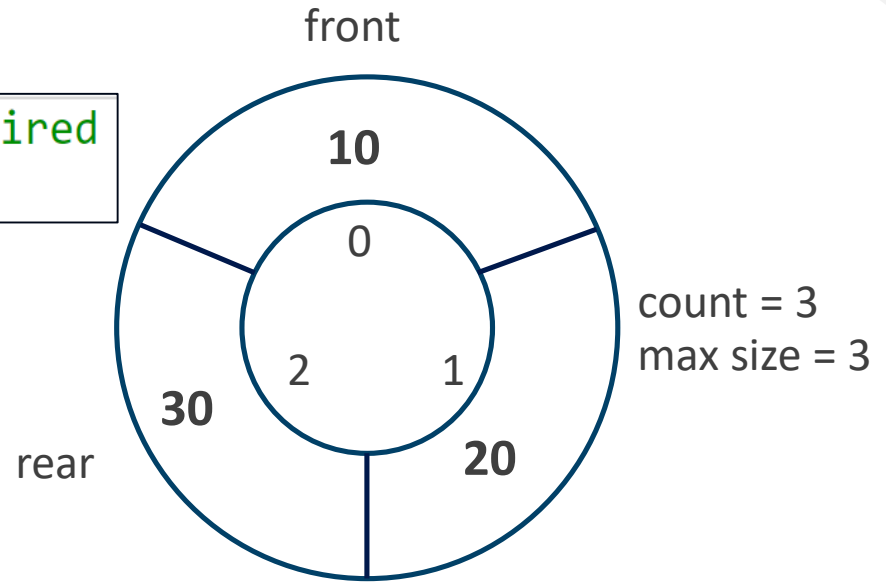


Fig 7-36 Empty circular queue

Circular Queue API

Empty circular queue

- This method verify the circular queue whether it is empty or not

```
1  def IsEmpty(self): #Check whether the queue is empty or not
2      return self.count == 0 #Not self.queueArr == []
```

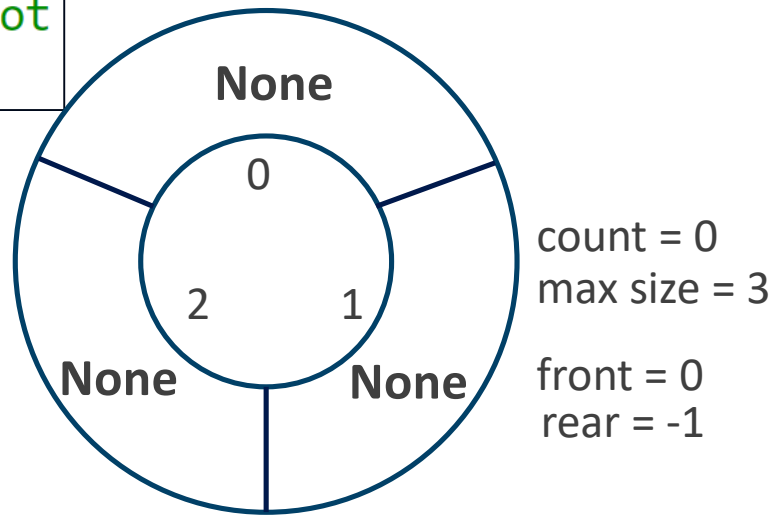


Fig 7-37 Full circular queue

Circular Queue API

Enqueue

- If we are at end and the element at the array head is free, wrap up to the beginning will be done.

```
1  def Enqueue(self, data): #Insert new element to queue rear
2    if self.IsFull() == False:
3      self.rear+=1
4      #Wrap to the beginning
5      self.rear = (self.rear)%self.maxSize
6      self.queueArr[self.rear] = data
7      self.count+=1 #Increase element counter
8    else:
9      print("Queue is Overflow")
```

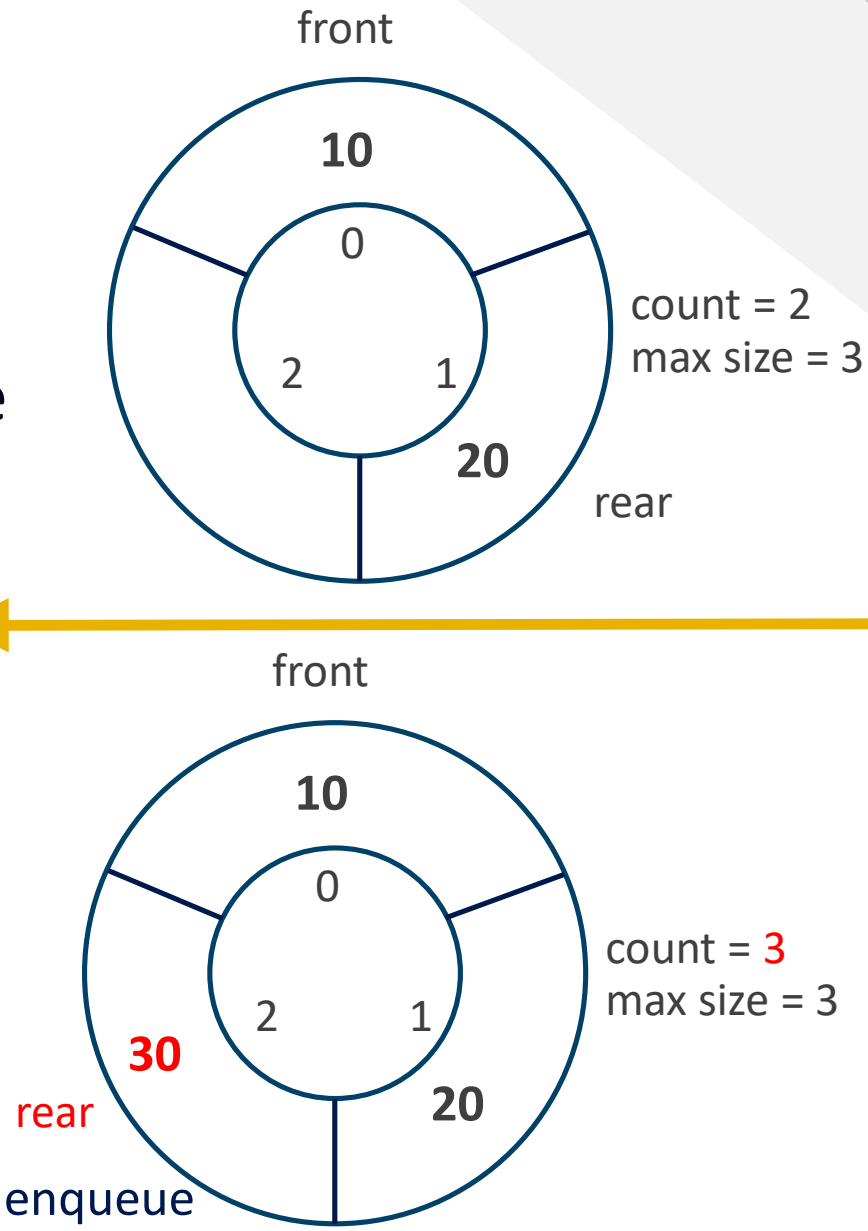


Fig 7-38 Circular queue enqueue

Circular Queue API

Dequeue

- If front index is at the last array index, updating it after dequeue will wrap around to the beginning.

```
1  def Dequeue(self): #Delete the front element
2    if self.IsEmpty()==False:
3      dequeueValue = self.queueArr[self.front]
4      self.queueArr[self.front] = "None"
5      #Wrap to the beginning
6      self.front = (self.front)%self.maxSize
7      self.front+=1
8      self.count-=1 #Decrease element counter
9    else:
10     print("Circular queue is empty")
11   return dequeueValue
```

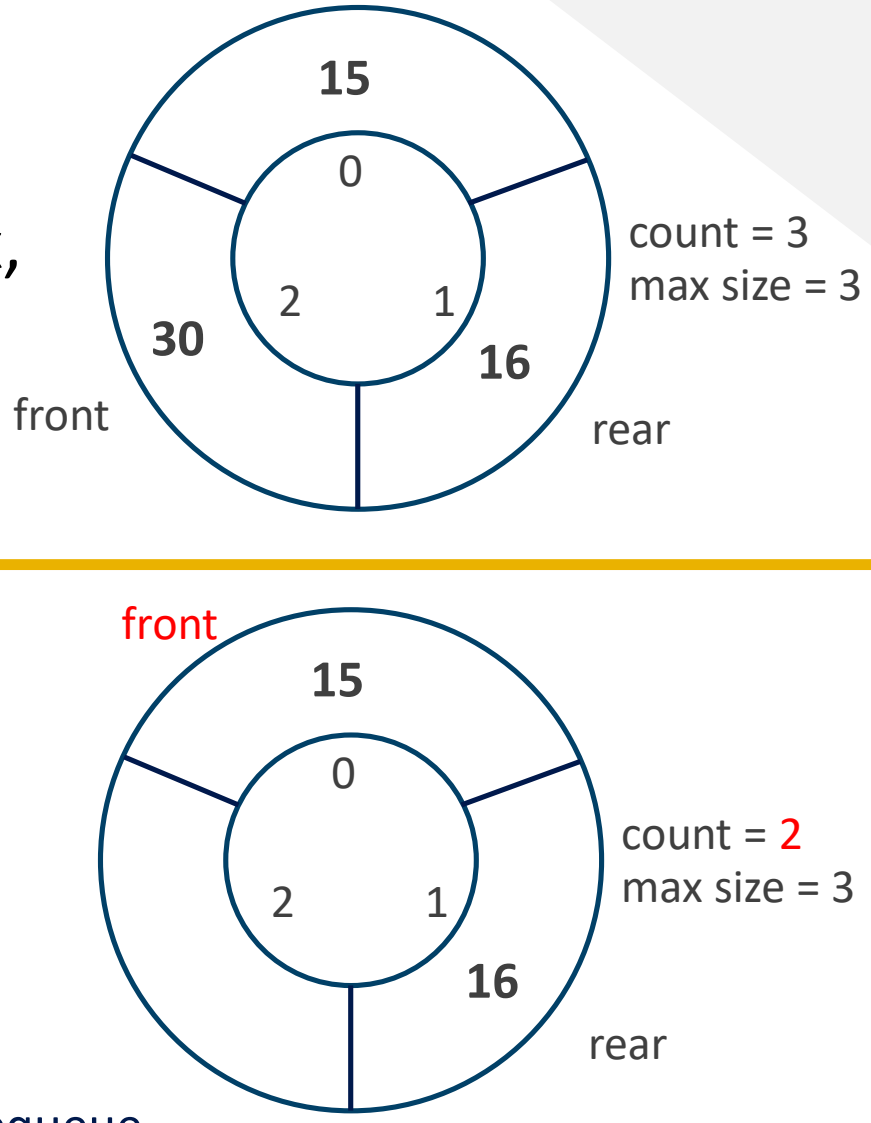


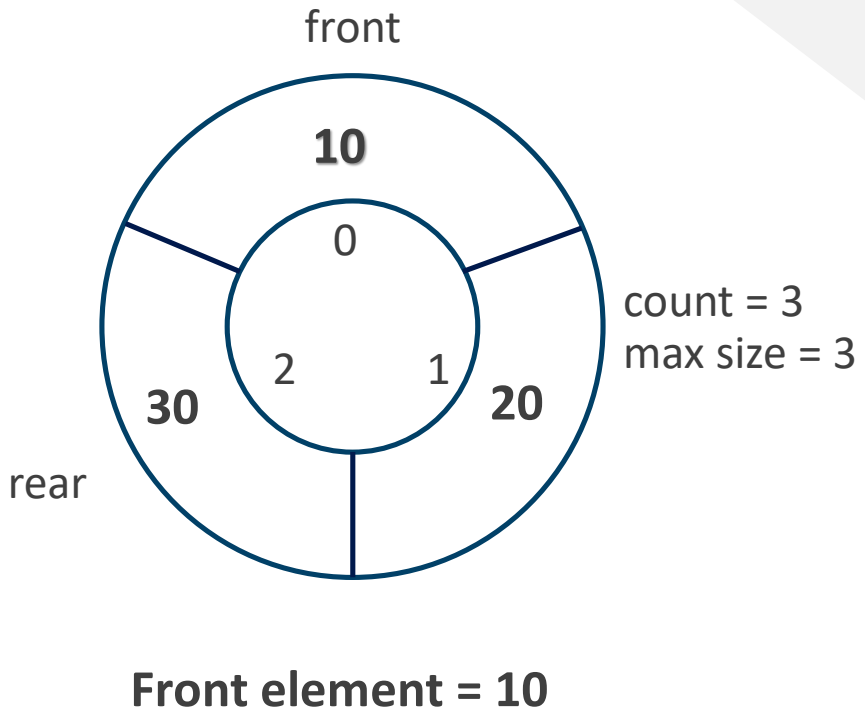
Fig 7-39 Circular queue Dequeue

Circular Queue API

Front

- Return the front element

```
1   def Front(self):
2     return self.queueArr[self.front]
```



Front element = 10

Fig 7-40 Circular queue front

Circular Queue API

Rear

- Return the rear element

```
1   def Rear(self):
2     return self.queueArr[self.rear]
```

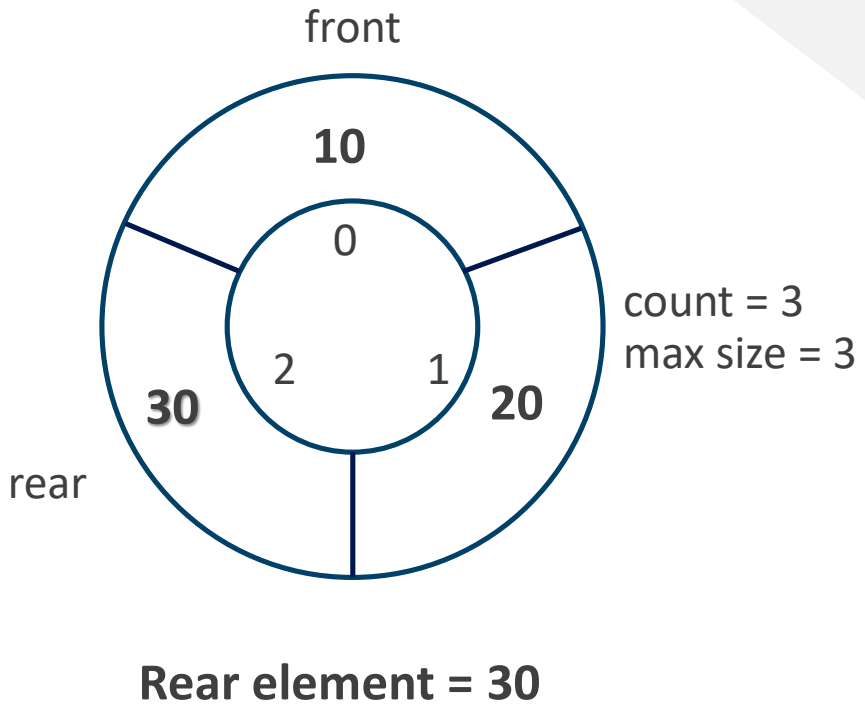


Fig 7-41 Circular queue rear

```
1  class QueueArray:
2    def __init__(self,maxSize): # Circular queue constructor
3      self.queueArr = []
4      self.front=0
5      self.rear=-1
6      self.count=0
7      self.maxSize= maxSize #Limit queue size
8      i=0
9      print("Circular queue - size :",self.maxSize)
10     while i<=maxSize-1:
11       #self.queueArr[i] = "None"
12       self.queueArr.insert(i, "None")
13       print(i, ".", self.queueArr[i])
14       i+=1
15
```

```
16  def IsFull(self): #Limit queue size, check size is required
17    return self.count == self.maxSize
18
19  def IsEmpty(self): #Check whether the queue is empty or not
20      return self.count == 0 #Not self.queueArr == []
21
22  def DisplayQueue(self): #Print entire queue elements
23    print("Circular queue contains:", self.QueueSize(), "elements")
24    i=0
25    while i<=len(self.queueArr)-1:
26      print(i, ".", self.queueArr[i])
27      i+=1
28
```

```
29  def Enqueue(self, data): #Insert new element to queue rear
30    if self.IsFull() == False:
31      self.rear+=1
32      #Wrap to the beginning
33      self.rear = (self.rear)%self.maxSize
34      self.queueArr[self.rear] = data
35      self.count+=1 #Increase element counter
36    else:
37      print("Queue is Overflow")
38
39  def Dequeue(self): #Delete the front element
40    if self.IsEmpty()==False:
41      dequeueValue = self.queueArr[self.front]
42      self.queueArr[self.front] = "None"
43      #Wrap to the beginning
44      self.front = (self.front)%self.maxSize
45      self.front+=1
46      self.count-=1 #Decrease element counter
47    else:
48      print("Circular queue is empty")
49    return dequeueValue
50
```

```
51    def Front(self):
52        return self.queueArr[self.front]
53
54    def Rear(self):
55        return self.queueArr[self.rear]
56
57    def QueueSize(self): #Return node counter
58        return self.count
```

```
59 #---
60 #Create a fix size queue array
61 array1 = QueueArray(3)
62 if array1.IsEmpty(): #Is queue array empty?
63   print("Circular queue is empty")
64 else:
65   print("Circular queue is not empty")
66 #Enqueue new nodes
67 array1.Enqueue(10)
68 array1.DisplayQueue()
69 array1.Enqueue(20)
70 array1.DisplayQueue()
71 array1.Enqueue(30)
72 array1.DisplayQueue()
73 array1.Enqueue(15)
74 array1.DisplayQueue()
```

```
#DeQueue and return the front element
print ("Dequeue element is : ",array1.Dequeue())
array1.DisplayQueue()
print ("Dequeue element is : ",array1.Dequeue())
array1.DisplayQueue()
#Enqueue new nodes
array1.Enqueue(15)
array1.DisplayQueue()
array1.Enqueue(16)
array1.DisplayQueue()
array1.Enqueue(17)
array1.DisplayQueue()
#Retrun the front element
print ("Front of queue value is : ",array1.Front())
#Retrun the rear element
print ("Rear of queue value is : ",array1.Rear())
```

```
Circular queue - size : 3
0 . None
1 . None
2 . None
Circular queue is empty
Circular queue contains: 1 elements
0 . 10
1 . None
2 . None
Circular queue contains: 2 elements
0 . 10
1 . 20
2 . None
Circular queue contains: 3 elements
0 . 10
1 . 20
2 . 30
Queue is Overflow
Circular queue contains: 3 elements
0 . 10
1 . 20
2 . 30
```

```
Dequeue element is :  10
Circular queue contains: 2 elements
0 . None
1 . 20
2 . 30
Dequeue element is :  20
Circular queue contains: 1 elements
0 . None
1 . None
2 . 30
Circular queue contains: 2 elements
0 . 15
1 . None
2 . 30
Circular queue contains: 3 elements
0 . 15
1 . 16
2 . 30
```

```
Queue is Overflow
Circular queue contains: 3 elements
0 . 15
1 . 16
2 . 30
Front of queue value is :  30
Rear of queue value is :  16
```

Circular Queue Methods' Performance

Analysis methods' peroformance of circular queue

Methods / Operations	Description	Big-O
Create a circular queue	Initial default values to circular queue	O(n)
Traverse queue	Update the array index to traverse every element of queue	O(n)
Empty / Full queue	It is only check whether circular is empty or not	O(1)
Enqueue	Append a new element to the end of circular queue	O(1)
Dequeue	Remove the last element of circular queue	O(1)
Front / Rear	Return a last element of circular queue	O(1)

Table 7-3 Analysis of circular queue operations

References

Texts | Integrated Development Environment (IDE)

[1] Data Structures and Algorithms in Python, Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, Willy & Sons Inc., 2013.

[2] Data Structures and Algorithms Using Python, Rance D. Necaise, John Winley & Sons, Inc., 2011.

[3] Problem Solving with Algorithms and Data Structures, Brad Minller and David Ranum, Python, 2013.

[4] Data Structures: A Pseudocode Approach with C++, Richard F. Gilberg and Behrouz A. Forouzan, Brooks/Cole, 2001.

[5] Problem Solving in Data Structures & Algorithms Using Python: Programming Interview Guide, 1st Edition, Hermant Jain, Thiftbooks, March 2017.

[6] https://trinket.io/features/python3

[7] http://interactivepython.org/courselib/static/pythonds/BasicDS/ImplementingaStackinPython.html