

## 06 Stacks and Queues

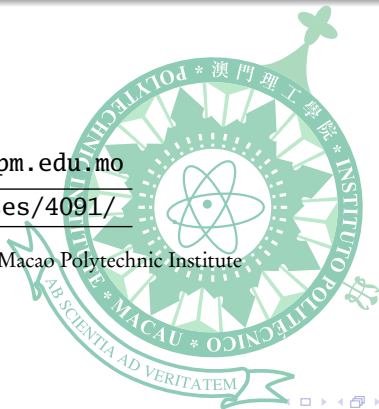
*Instructor* : Ke Wei [ 柯韋 ]

➡ A319    ☎ Ext. 6452    ✉ wke@ipm.edu.mo

<https://canvas.ipm.edu.mo/courses/4091/>


Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

February 10, 2022



# Outline

- 1 Stacks
- 2 Array-Based Stacks
- 3 Applications of Stacks
- 4 Queues
- 5 Linked List-Based Queues
- 6 Circular Buffer-Based Queues

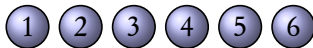
 *Textbook §6.1 – 6.2, §7.1.1 – 7.1.2.*

# Stacks and LIFOs

- For a singly linked lists, if we keep inserting elements at the head position, and then keep removing the head elements. We have the following property:

*The last element inserted to the list is the first element deleted.*

- This property is called “Last-In, First-Out”, or LIFO.
- Another common name for this kind of structure is “stack”.
- Stacks are abstract, any collections of elements that have the LIFO property can be regarded as stacks, or LIFOs.



# Stacks and LIFOs

- For a singly linked lists, if we keep inserting elements at the head position, and then keep removing the head elements. We have the following property:

*The last element inserted to the list is the first element deleted.*

- This property is called “Last-In, First-Out”, or LIFO.
- Another common name for this kind of structure is “stack”.
- Stacks are abstract, any collections of elements that have the LIFO property can be regarded as stacks, or LIFOs.

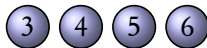


# Stacks and LIFOs

- For a singly linked lists, if we keep inserting elements at the head position, and then keep removing the head elements. We have the following property:

*The last element inserted to the list is the first element deleted.*

- This property is called “Last-In, First-Out”, or LIFO.
- Another common name for this kind of structure is “stack”.
- Stacks are abstract, any collections of elements that have the LIFO property can be regarded as stacks, or LIFOs.

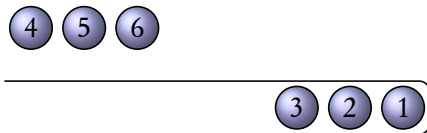


# Stacks and LIFOs

- For a singly linked lists, if we keep inserting elements at the head position, and then keep removing the head elements. We have the following property:

*The last element inserted to the list is the first element deleted.*

- This property is called “Last-In, First-Out”, or LIFO.
- Another common name for this kind of structure is “stack”.
- Stacks are abstract, any collections of elements that have the LIFO property can be regarded as stacks, or LIFOs.

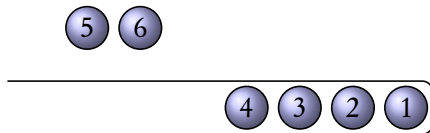


# Stacks and LIFOs

- For a singly linked lists, if we keep inserting elements at the head position, and then keep removing the head elements. We have the following property:

*The last element inserted to the list is the first element deleted.*

- This property is called “Last-In, First-Out”, or LIFO.
- Another common name for this kind of structure is “stack”.
- Stacks are abstract, any collections of elements that have the LIFO property can be regarded as stacks, or LIFOs.

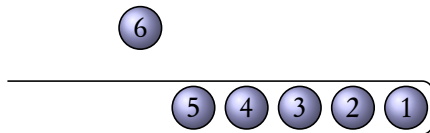


# Stacks and LIFOs

- For a singly linked lists, if we keep inserting elements at the head position, and then keep removing the head elements. We have the following property:

*The last element inserted to the list is the first element deleted.*

- This property is called “Last-In, First-Out”, or LIFO.
- Another common name for this kind of structure is “stack”.
- Stacks are abstract, any collections of elements that have the LIFO property can be regarded as stacks, or LIFOs.



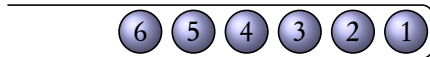


# Stacks and LIFOs

- For a singly linked lists, if we keep inserting elements at the head position, and then keep removing the head elements. We have the following property:

*The last element inserted to the list is the first element deleted.*

- This property is called “Last-In, First-Out”, or LIFO.
- Another common name for this kind of structure is “stack”.
- Stacks are abstract, any collections of elements that have the LIFO property can be regarded as stacks, or LIFOs.

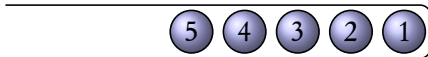


# Stacks and LIFOs

- For a singly linked lists, if we keep inserting elements at the head position, and then keep removing the head elements. We have the following property:

*The last element inserted to the list is the first element deleted.*

- This property is called “Last-In, First-Out”, or LIFO.
- Another common name for this kind of structure is “stack”.
- Stacks are abstract, any collections of elements that have the LIFO property can be regarded as stacks, or LIFOs.

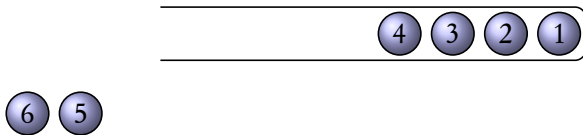


# Stacks and LIFOs

- For a singly linked lists, if we keep inserting elements at the head position, and then keep removing the head elements. We have the following property:

*The last element inserted to the list is the first element deleted.*

- This property is called “Last-In, First-Out”, or LIFO.
- Another common name for this kind of structure is “stack”.
- Stacks are abstract, any collections of elements that have the LIFO property can be regarded as stacks, or LIFOs.

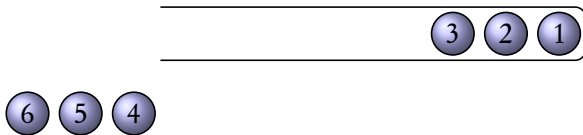


# Stacks and LIFOs

- For a singly linked lists, if we keep inserting elements at the head position, and then keep removing the head elements. We have the following property:

*The last element inserted to the list is the first element deleted.*

- This property is called “Last-In, First-Out”, or LIFO.
- Another common name for this kind of structure is “stack”.
- Stacks are abstract, any collections of elements that have the LIFO property can be regarded as stacks, or LIFOs.

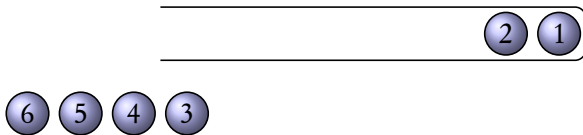


# Stacks and LIFOs

- For a singly linked lists, if we keep inserting elements at the head position, and then keep removing the head elements. We have the following property:

*The last element inserted to the list is the first element deleted.*

- This property is called “Last-In, First-Out”, or LIFO.
- Another common name for this kind of structure is “stack”.
- Stacks are abstract, any collections of elements that have the LIFO property can be regarded as stacks, or LIFOs.

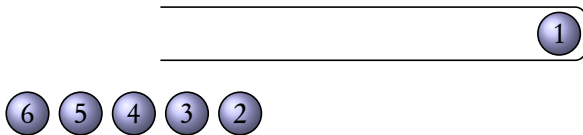


# Stacks and LIFOs

- For a singly linked lists, if we keep inserting elements at the head position, and then keep removing the head elements. We have the following property:

*The last element inserted to the list is the first element deleted.*

- This property is called “Last-In, First-Out”, or LIFO.
- Another common name for this kind of structure is “stack”.
- Stacks are abstract, any collections of elements that have the LIFO property can be regarded as stacks, or LIFOs.

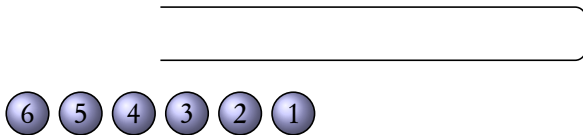


# Stacks and LIFOs

- For a singly linked lists, if we keep inserting elements at the head position, and then keep removing the head elements. We have the following property:

*The last element inserted to the list is the first element deleted.*

- This property is called “Last-In, First-Out”, or LIFO.
- Another common name for this kind of structure is “stack”.
- Stacks are abstract, any collections of elements that have the LIFO property can be regarded as stacks, or LIFOs.



# The *Stack* ADT

Formally, a stack is an *abstract data type* (ADT) such that an instance supports the following operations.

- $push(self, x)$  — add element  $x$  to the top of the stack.
- $pop(self)$  — remove and return the top element from the stack, an error occurs if the stack is empty.
- $top(self)$  — return a reference to the top element of the stack, without removing it, an error occurs if the stack is empty.
- $__bool__(self)$  — return **True** if the stack contains some elements, **False** if empty.

The singly linked list *LnLs* supports all the listed operations, thus, can be used directly as a stack.



# Array-Based Stacks

- We can also use an array-based list as the storage of a stack, eliminating the node creation at each “push” for a linked list.
- System stacks manipulated by processors are implemented as even a fixed length array to achieve high efficiency.
- Since an array-based list can only be extended at the end, we push new elements by appending.
- Accordingly, we pop an element from the stack by deleting and returning the last item of the list.

# Array-Based Stacks — Code

```
1 class AStack:
2     def __init__(self):
3         self.a = []
4     def __bool__(self):
5         return self.a != []
6     def top(self):
7         return self.a[-1]
8     def push(self, x):
9         self.a.append(x)
10    def pop(self):
11        x = self.top()
12        del self.a[-1]
13        return x
```

# Applications of Stacks

Direct applications:

- Page-visited history in a Web browser.
- Undo sequence in a text editor.
- Chain of method calls in the Java Virtual Machine.

Indirect applications:

- Auxiliary data structure for algorithms.
- Component of other data structures.

# Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “]”.

Correct : ( )(( )){([ ( ) ])}{ }

Correct : ((( ))(( ))){([ ( ) ])}{ }

Incorrect : )(( )){([ ( ) ])}{ }

Incorrect : ({ [ ] ) }

Incorrect : ([ ( ) ])

- We use a stack to store opening symbols. We scan the string, and for each character  $c$ ,
  - if  $c$  is an opening symbol, we push it onto the stack.
  - if  $c$  is a closing symbol, we pop the stack and see if it matches  $c$ . If we cannot pop the stack or the symbol popped out does not match, we declare a mismatch.
- When the scanning completes without mismatches, and the stack is finally empty, then the string is correct.

# Parentheses Matching — Code

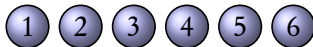
---

```
1 def paren_match(s):
2     st = Stack()
3     for c in s:
4         if c in ('(', '[', '{'):
5             st.push(c)
6         elif c in (')', ']', '}'):
7             if not st or st.pop()+c not in ('()', '[]', '{}'):
8                 return False
9     return not st
```

---

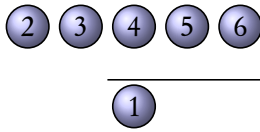
# Queues and FIFOs

- Queues are linear structures that insert elements at one end and delete elements at the other end.
- The first element inserted to a queue is deleted first.
- Insertion at the rear end is called “enqueue” or “push back”, deletion at the front end is called “dequeue” or “pop”.
- A queue is also called a FIFO (First-In, First-Out).
- Queues are also abstract, any collections of elements that have the FIFO property can be regarded as queues, or FIFOs.



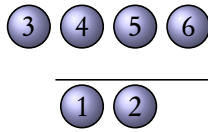
# Queues and FIFOs

- Queues are linear structures that insert elements at one end and delete elements at the other end.
- The first element inserted to a queue is deleted first.
- Insertion at the rear end is called “enqueue” or “push back”, deletion at the front end is called “dequeue” or “pop”.
- A queue is also called a FIFO (First-In, First-Out).
- Queues are also abstract, any collections of elements that have the FIFO property can be regarded as queues, or FIFOs.



# Queues and FIFOs

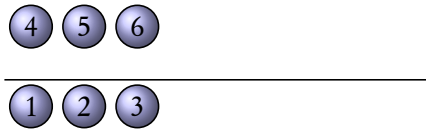
- Queues are linear structures that insert elements at one end and delete elements at the other end.
- The first element inserted to a queue is deleted first.
- Insertion at the rear end is called “enqueue” or “push back”, deletion at the front end is called “dequeue” or “pop”.
- A queue is also called a FIFO (First-In, First-Out).
- Queues are also abstract, any collections of elements that have the FIFO property can be regarded as queues, or FIFOs.





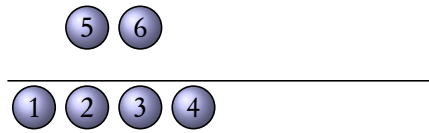
# Queues and FIFOs

- Queues are linear structures that insert elements at one end and delete elements at the other end.
- The first element inserted to a queue is deleted first.
- Insertion at the rear end is called “enqueue” or “push back”, deletion at the front end is called “dequeue” or “pop”.
- A queue is also called a FIFO (First-In, First-Out).
- Queues are also abstract, any collections of elements that have the FIFO property can be regarded as queues, or FIFOs.



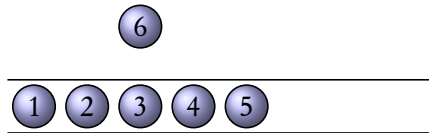
# Queues and FIFOs

- Queues are linear structures that insert elements at one end and delete elements at the other end.
- The first element inserted to a queue is deleted first.
- Insertion at the rear end is called “enqueue” or “push back”, deletion at the front end is called “dequeue” or “pop”.
- A queue is also called a FIFO (First-In, First-Out).
- Queues are also abstract, any collections of elements that have the FIFO property can be regarded as queues, or FIFOs.



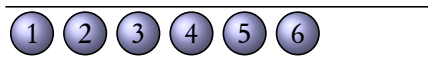
# Queues and FIFOs

- Queues are linear structures that insert elements at one end and delete elements at the other end.
- The first element inserted to a queue is deleted first.
- Insertion at the rear end is called “enqueue” or “push back”, deletion at the front end is called “dequeue” or “pop”.
- A queue is also called a FIFO (First-In, First-Out).
- Queues are also abstract, any collections of elements that have the FIFO property can be regarded as queues, or FIFOs.



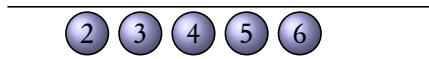
# Queues and FIFOs

- Queues are linear structures that insert elements at one end and delete elements at the other end.
- The first element inserted to a queue is deleted first.
- Insertion at the rear end is called “enqueue” or “push back”, deletion at the front end is called “dequeue” or “pop”.
- A queue is also called a FIFO (First-In, First-Out).
- Queues are also abstract, any collections of elements that have the FIFO property can be regarded as queues, or FIFOs.



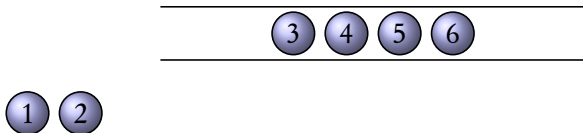
# Queues and FIFOs

- Queues are linear structures that insert elements at one end and delete elements at the other end.
- The first element inserted to a queue is deleted first.
- Insertion at the rear end is called “enqueue” or “push back”, deletion at the front end is called “dequeue” or “pop”.
- A queue is also called a FIFO (First-In, First-Out).
- Queues are also abstract, any collections of elements that have the FIFO property can be regarded as queues, or FIFOs.



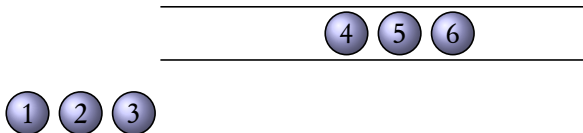
# Queues and FIFOs

- Queues are linear structures that insert elements at one end and delete elements at the other end.
- The first element inserted to a queue is deleted first.
- Insertion at the rear end is called “enqueue” or “push back”, deletion at the front end is called “dequeue” or “pop”.
- A queue is also called a FIFO (First-In, First-Out).
- Queues are also abstract, any collections of elements that have the FIFO property can be regarded as queues, or FIFOs.



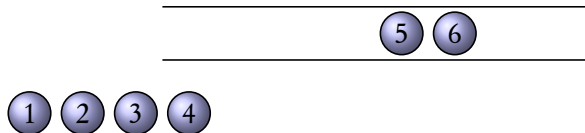
# Queues and FIFOs

- Queues are linear structures that insert elements at one end and delete elements at the other end.
- The first element inserted to a queue is deleted first.
- Insertion at the rear end is called “enqueue” or “push back”, deletion at the front end is called “dequeue” or “pop”.
- A queue is also called a FIFO (First-In, First-Out).
- Queues are also abstract, any collections of elements that have the FIFO property can be regarded as queues, or FIFOs.



# Queues and FIFOs

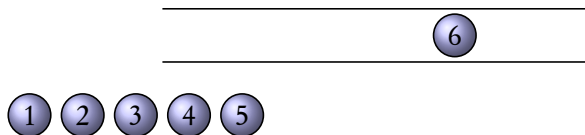
- Queues are linear structures that insert elements at one end and delete elements at the other end.
- The first element inserted to a queue is deleted first.
- Insertion at the rear end is called “enqueue” or “push back”, deletion at the front end is called “dequeue” or “pop”.
- A queue is also called a FIFO (First-In, First-Out).
- Queues are also abstract, any collections of elements that have the FIFO property can be regarded as queues, or FIFOs.





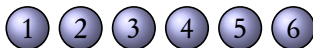
# Queues and FIFOs

- Queues are linear structures that insert elements at one end and delete elements at the other end.
- The first element inserted to a queue is deleted first.
- Insertion at the rear end is called “enqueue” or “push back”, deletion at the front end is called “dequeue” or “pop”.
- A queue is also called a FIFO (First-In, First-Out).
- Queues are also abstract, any collections of elements that have the FIFO property can be regarded as queues, or FIFOs.



# Queues and FIFOs

- Queues are linear structures that insert elements at one end and delete elements at the other end.
- The first element inserted to a queue is deleted first.
- Insertion at the rear end is called “enqueue” or “push back”, deletion at the front end is called “dequeue” or “pop”.
- A queue is also called a FIFO (First-In, First-Out).
- Queues are also abstract, any collections of elements that have the FIFO property can be regarded as queues, or FIFOs.



# The Queue ADT

Formally, a queue is an ADT such that an instance supports the following operations.

- *push\_back*(*self*, *x*) — add element *x* to the end of the queue.
- *pop*(*self*) — remove and return the first element from the queue, an error occurs if the queue is empty.
- *top*(*self*) — return a reference to the first element of the queue, without removing it, an error occurs if the queue is empty.
- *\_\_bool\_\_*(*self*) — return **True** if the queue contains some elements, **False** if empty.

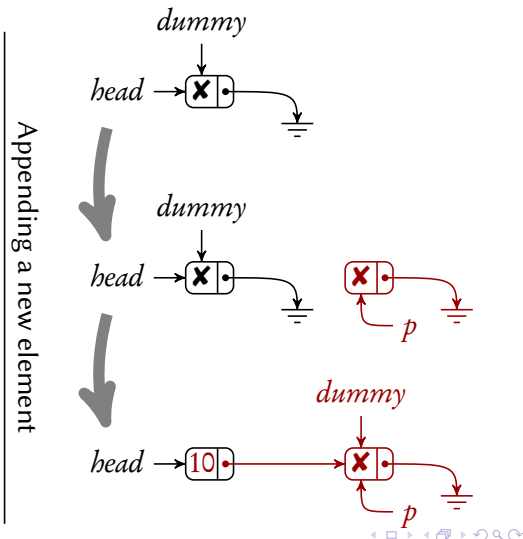
# Applications of Queues

Direct applications:

- Waiting lists, bureaucracy.
- Access to shared resources (e.g., printer).
- Multiprogramming.

Indirect applications

- Auxiliary data structure for algorithms.
- Component of other data structures.



# Linked List-Based Queues

- We need to append a node next to the last node in order to *push\_back*.
- A reference to the last node must be recorded.
- However, when the linked list is empty, we do not have the last node.
- To unify the handling of the two cases, we introduce a *dummy* node as the last node.
- When we *push\_back*, we simply put the new element into the old dummy node, and link it to a new dummy node.

---

```

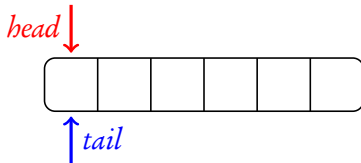
1 class LQueue:
2     def __init__(self):
3         self.head = self.dummy = Node(None, None)
4     def __bool__(self):
5         return self.head is not self.dummy
6     def push_back(self, x):
7         self.dummy.elm, self.dummy.next = x, Node(None, None)
8         self.dummy = self.dummy.next

```

---

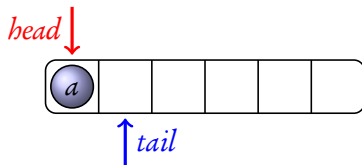
# Circular Buffer-Based Queues

- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.



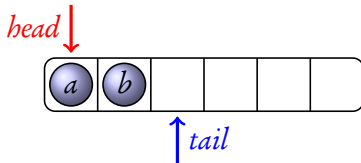
# Circular Buffer-Based Queues

- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.



# Circular Buffer-Based Queues

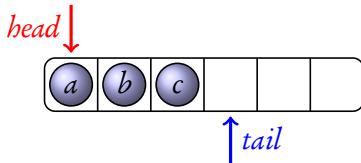
- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.





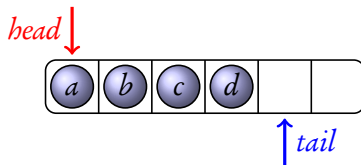
# Circular Buffer-Based Queues

- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.



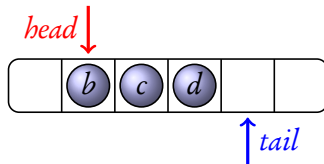
# Circular Buffer-Based Queues

- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.



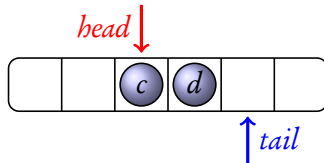
# Circular Buffer-Based Queues

- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.



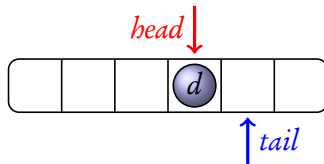
# Circular Buffer-Based Queues

- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.



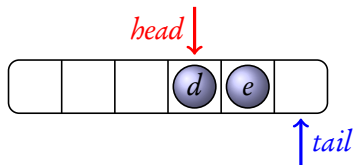
# Circular Buffer-Based Queues

- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.



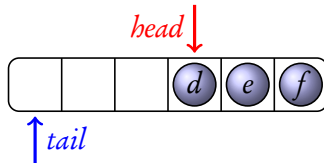
# Circular Buffer-Based Queues

- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.



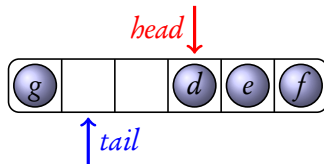
# Circular Buffer-Based Queues

- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.



# Circular Buffer-Based Queues

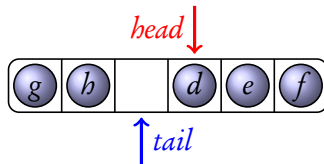
- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.





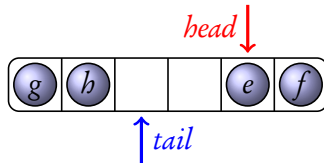
# Circular Buffer-Based Queues

- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.



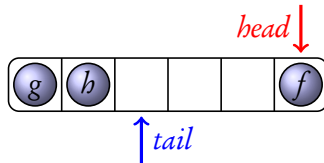
# Circular Buffer-Based Queues

- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.



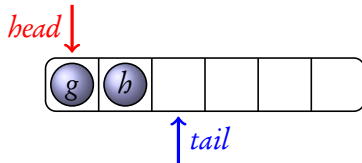
# Circular Buffer-Based Queues

- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.



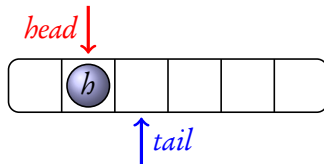
# Circular Buffer-Based Queues

- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.



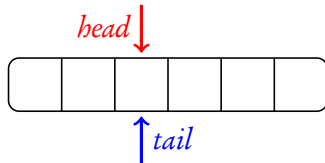
# Circular Buffer-Based Queues

- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.



# Circular Buffer-Based Queues

- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.



# Full and Empty Circular Buffers

- How many elements can be held in a circular buffer of size  $n$ ?
- The number of elements in a circular buffer can be computed using a modular subtraction:

$$(tail - head) \% n.$$

- The range of the above expression is from 0 to  $n - 1$ . Therefore, there are at most  $n - 1$  elements that can be held without additional information.
- How to wrap?

$$head \leftarrow (head + 1) \% n.$$

- How to detect if a circular buffer is empty or full?
  - Obviously, if  $head$  equals  $tail$ , there is no element.
  - If  $(tail + 1) \% n$  reaches  $head$ , the buffer is full.
  - When the queue is full, we can extend the underlying list at  $tail$ , and adjust  $head$  accordingly.

# Circular Buffer-Based Queues — Code

```
1 class AQueue:
2     def __init__(self):
3         self.a = [None]*16
4         self.n = len(self.a)
5         self.head = self.tail = 0
6
7     def __bool__(self):
8         return self.head != self.tail
9
10    def __len__(self):
11        return (self.tail-self.head)%self.n
12
13    def __iter__(self):
14        yield from (self.a[(self.head+i)%self.n] for i in range(len(self)))
```



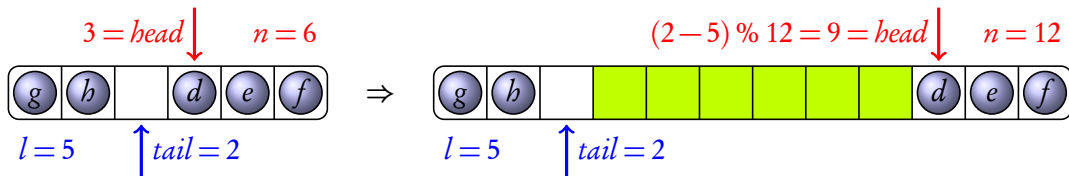
## Circular Buffer-Based Queues — Code (2)

---

```
16 def top(self):
17     if not self:
18         raise IndexError
19     return self.a[self.head]
20
21 def pop(self):
22     x = self.top()
23     self.a[self.head] = None
24     self.head = (self.head+1)%self.n
25     return x
```

---

## Circular Buffer-Based Queues — Code (3)



```

26 def push_back(self, x):
27     l = len(self)
28     if l+1 == self.n: # back-end list is full
29         self.a[self.tail:self.tail] = [None]*self.n
30         self.n = len(self.a)
31         self.head = (self.tail-l)%self.n
32
33     self.a[self.tail] = x
34     self.tail = (self.tail+1)%self.n

```

