Datenstrukturen: Queue (Warteschlange)

Prüfungsvorbereitung

Aufgabe 1

Beschreibe möglichst viele strukturellen Merkmale der Datenstruktur *Queue*?

Aufgabe 1

Es handelt sich um eine Sammlung von Elementen, in der

- die Reihenfolge der Elemente wesentlich ist,
- Elemente wiederholt vorkommen können,
- das zuerst eingefügte Element auch als erstes entnommen wird (Kurzform: First In – First Out oder noch kürzer: FIFO)

Aufgabe 2

Beschreibe die typischen Operationen der Datenstruktur *Queue*.

Aufgabe 2

- Eine leere Queue erzeugen: `q = Queue()`
- Ein Element der Queue hinzufügen: `q.enqueue(item)`
- Ein Element der Queue entnehmen: `item = q.dequeue()`
- Testen, ob die Queue leer ist: `q.is_empty()`
- Anzahl der Elemente der Queue zurückgeben: `q.size()`
- Die Queue löschen: `q.clear()`

Aufgabe 3

Gib zwei unterschiedliche Anwendungen der Datenstruktur *Queue* an.

Aufgabe 3

- Als Datenpuffer für externe Ein- und Ausgabegeräte wie Tastaturen, Mäuse oder Drucker
- Als Datenpuffer für die Interprozesskommunikation.
  Interprozesskommunikation bedeutet, dass zwei Prozesse über ein gemeinsames Medium (meist ein gemeinsamer Speicherbereich) Daten austauschen.
- *Prozess-Scheduler*
  Ein *Prozess-Scheduler* bezeichnet einen Algorithmus (oder ein Programm), der regelt, wie und in welcher Reihenfolge die auszuführenden Programme vom Prozessor ausgeführt werden.
- Als Datenstruktur für Algorithmen wie die Breitensuche in Graphen.

Aufgabe 4

Warum eignen sich Python-Listen nur bedingt zur Implementierung von Queues?

Aufgabe 4

Python kann ein Element *am Ende* einer Liste in $O(1)$ hinzufügen, was schnell ist. Hingegen benötigt Python $O(n)$, um Elemente vom *am Anfang* der Liste zu entferen (weil die Elemente rechts davon „verschoben“ werden müssen.

Würde man die Elemente umgekehrt am Anfang der Liste einfügen, hätte dies dies auch wieder eine Komplexität von $O(n)$, während das Entfernen am Listenende wieder in $O(1)$ erfolgt.

Somit eignet sich die listenbasierte Implementierung nicht für Queues, die viele Daten schnell verarbeiten müssen.

Aufgabe 5

Welche Ausgaben macht der folgende Python-Code?

```
1  from queue import Queue
2  q = Queue()
3  q.enqueue(7)
4  q.enqueue(2)
5  q.enqueue(3)
6  q.enqueue(5)
7  x = q.dequeue()
8  q.enqueue(1)
9  q.enqueue(4)
10 y = q.dequeue()
11 print(x)
12 print(y)
13 print(q.size())
14 q.enqueue(q.dequeue())
15 q.enqueue(q.dequeue())
16 z = q.dequeue()
17 print(z)
```

Aufgabe 5

```
1  from queue import Queue
2  q = Queue()       # -> ->
3  q.enqueue(7)      # -> 7 ->
4  q.enqueue(2)      # -> 2, 7 ->
5  q.enqueue(3)      # -> 3, 2, 7 ->
6  q.enqueue(5)      # -> 5, 3, 2, 7 ->
7  x = q.dequeue() # -> 5, 3, 2 ->  (x=7)
8  q.enqueue(1)      # -> 1, 5, 3, 2 ->
9  q.enqueue(4)      # -> 4, 1, 5, 3, 2 ->
10 y = q.dequeue() # -> 4, 1, 5, 3 -> (y=2)
11 print(x)          # Ausgabe: 7
12 print(y)          # Ausgabe: 2
13 print(q.size()) # Ausgabe: 4
14 q.enqueue(q.dequeue()) # -> 3, 4, 1, 5 ->
15 q.enqueue(q.dequeue()) # -> 5, 3, 4, 1 ->
16 z = q.dequeue()          # -> 5, 3, 4 -> (z=1)
17 print(z)          # Ausgabe: 1
```

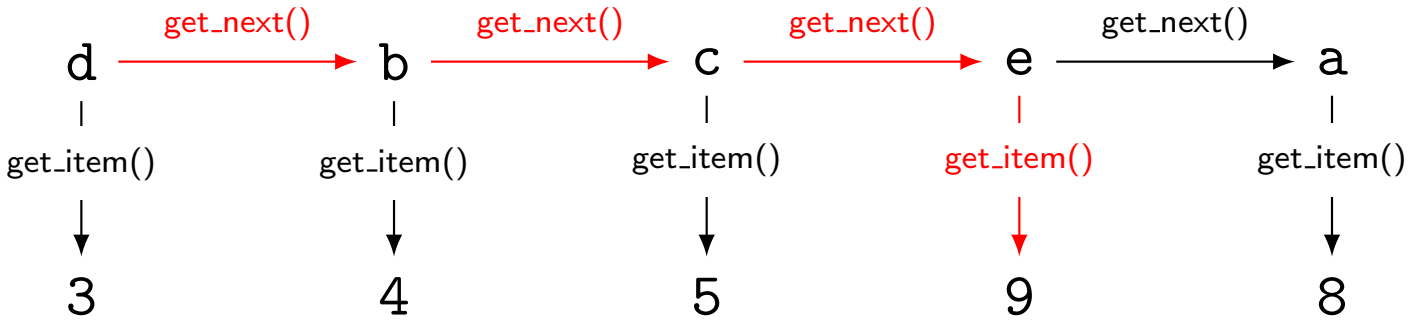
Aufgabe 6

Welche Ausgabe macht der folgende Code, der Node-Objekte mit einem Zeiger auf Nachbarknoten verküpft?

```
1  class Node:
2
3      def __init__(self, item, next = None):
4          self.item = item
5          self.next = next
6
7      def get_item(self):
8          return self.item
9
10     def get_next(self):
11         return self.next
```

```
13  c = Node(5)
14  a = Node(7)
15  d = Node(3)
16  e = Node(9)
17  b = Node(4)
18
19  e.next = a
20  d.next = b
21  c.next = e
22  b.next = c
23
24  print(d.get_next().get_next().get_next().get_item())
```

Aufgabe 6



Ausgabe: 9

Aufgabe 7

Vervollständige das Zeigerdiagram der Queue `q` anhand des Speicherabbildes, das noch andere Daten enthält. Ein Knotenobjekt besteht jeweils aus zwei aufeinanderfolgenden Feldern, wobei im ersten Feld der Wert und im zweiten die Referenz auf das erste Feld des nächsten Knoten steht. `00` steht für `None`. Welche Daten befinden sich in welcher Reihenfolge in der Queue?

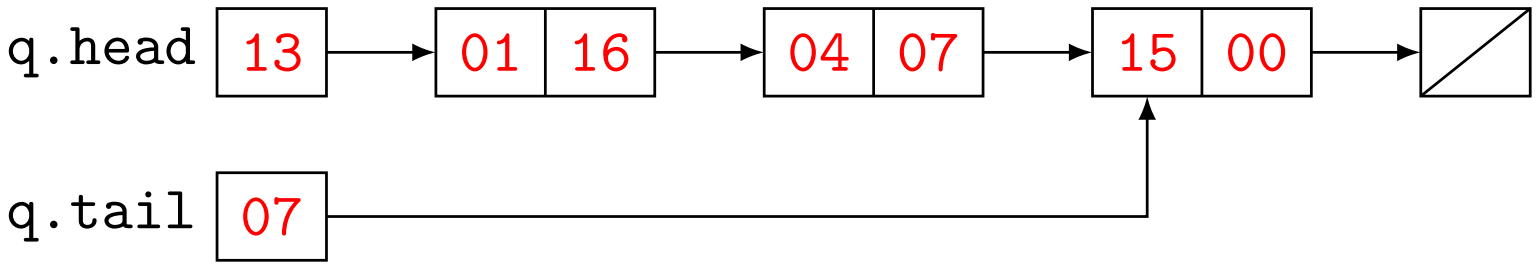
	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9
0.		06	00	08	16	15	00	15	00	09
1.	15	11	18	01	16	06	04	07	16	00

`q.head` ☐

`q.tail` ☐

Aufgabe 7

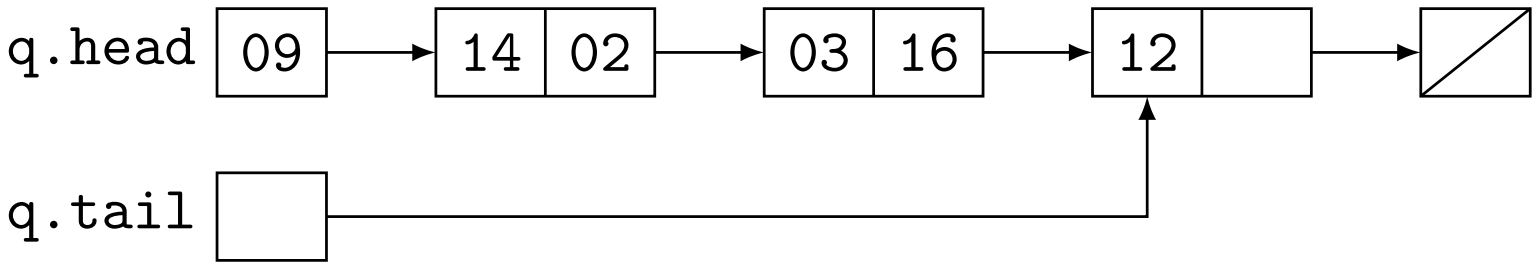
	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9
0.	/	06	00	08	16	15	00	15	00	09
1.	15	11	18	01	16	06	04	07	16	00



Daten: $\rightarrow$ 15 04 01 $\rightarrow$

Aufgabe 8

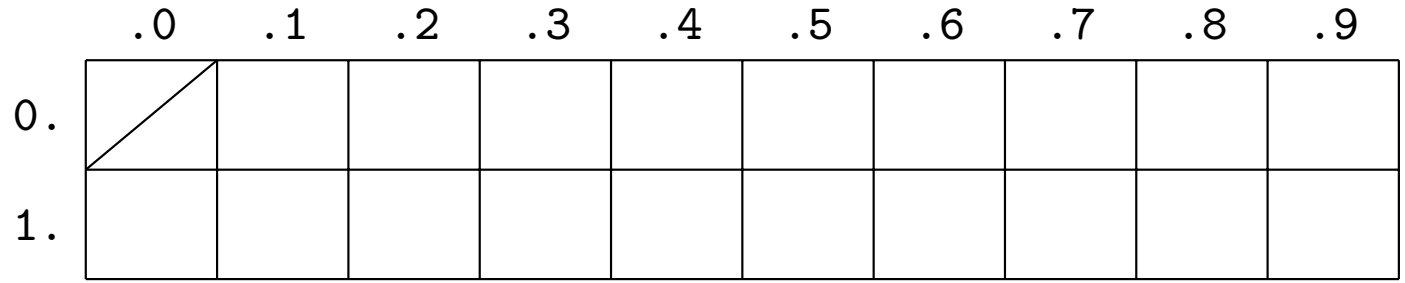
Gegeben ist folgendes Zeigerdiagramm einer Queue q.



(a) Trage die passenden Werte in die leeren Felder ein.

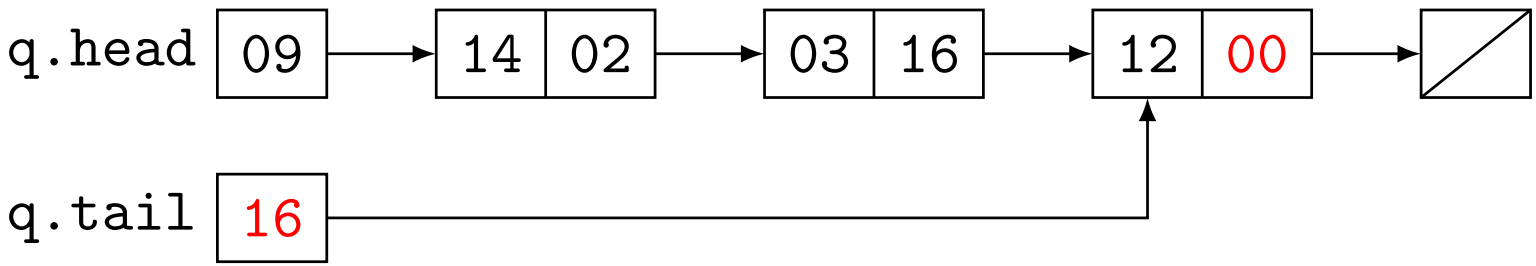
(b) Skizziere das Zeigerdiagramm nach einer dequeue-Operation auf q.

(c) Erstelle ein Speicherabbild der Queue q, nachdem (b) ausgeführt wurde. Berücksichtige auch den verwaisten Knoten.



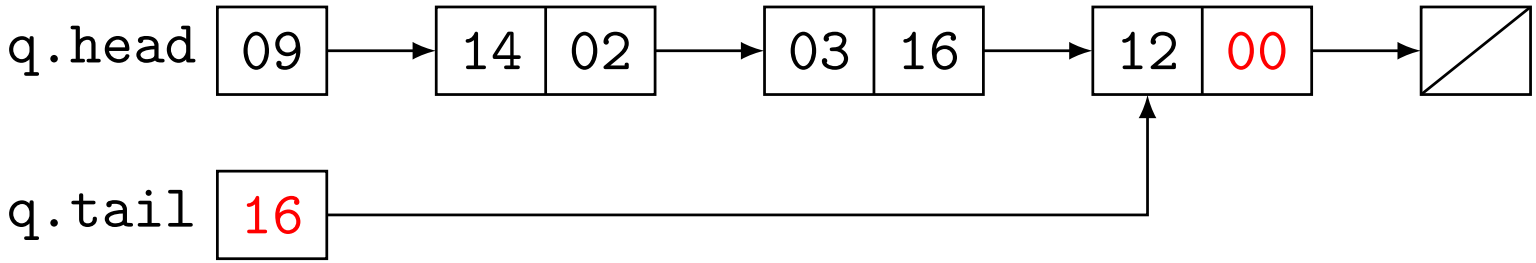
Aufgabe 8

(a)

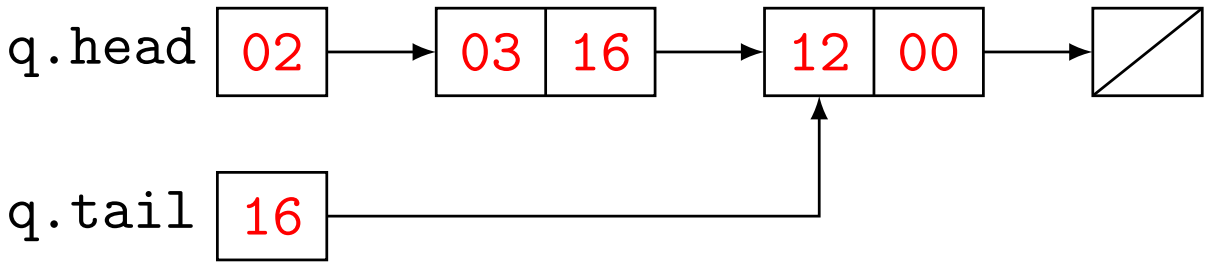


Aufgabe 8

(a)

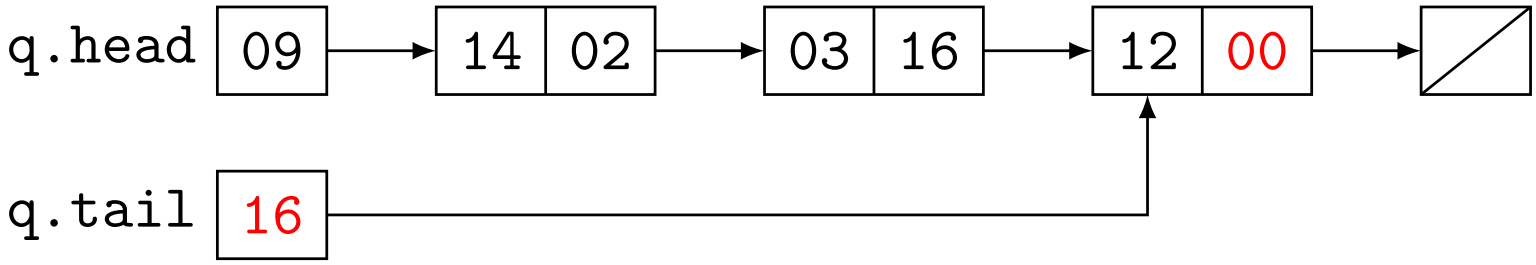


(b)

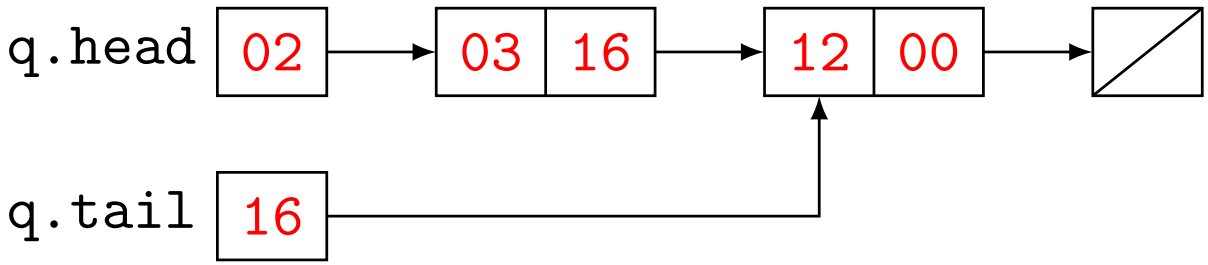


Aufgabe 8

(a)



(b)



(c)

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9
0.			03	16						14
1.	02						12	00		

Aufgabe 9

(a) Implementiere die Methoden `queue(item)` und `dequeue()` der Klasse `Queue` in Python auf der Basis einer Liste.

Hinweis: Es gibt mehrere korrekte Lösungen aber es genügt, eine anzugeben.

```
class Queue:

    def __init__(self):
        self.data=[]
```

(b) Gib die Laufzeitkomplexität der von dir in Python implementierten Methoden an, wenn die aktuelle Queue n Elemente enthält.

Aufgabe 9

(a)

```
class Queue:

    def __init__(self):
        self.data=[]

    def enqueue(self,item):
        self.data.append(item)

    def dequeue(self)   :
        return self.data.pop(0):
```

(b) `q.enqueue(item)`: $O(1)$

Kosten für das Einfügen eines Elements *am Listenende*.

(b) `q.enqueue(item)`: $O(1)$

Kosten für das Einfügen eines Elements *am Listenende*.

`q.dequeue()`: $O(n)$

Kosten für das Entfernen eines Elements *am Listenanfang*.