Gruppe 23
Salah Atallah, 414867
Mohammed Al-Laktah, 419664

Aufgabe 3

$$\sum_{i=1}^{n} i^2 = \frac{n \cdot (n+1) \cdot (2 \cdot n + 1)}{6}$$

I - IA)

$$\begin{aligned}
\sum_{i=1}^{n} i^2 &= \frac{n \cdot (n+1) \cdot (2 \cdot n + 1)}{6} \\
\sum_{i=1}^{1} i^2 &= \frac{1 \cdot (1+1) \cdot (2 \cdot 1 + 1)}{6} \\
1^2 &= \frac{1 \cdot (2) \cdot (2 \cdot 2)}{6} \\
1 &= \frac{2 \cdot (4)}{6} \\
1 &= \frac{6}{6} \\
1 &= 1
\end{aligned}$$

II - IV)

$$\sum_{i=1}^{n} i^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

III - IS)

$$z.z. \sum_{i=1}^{(n+1)} i^2 = \frac{(n+1) \cdot ((n+1)+1) \cdot (2 \cdot (n+1)+1)}{6}$$

IV)

$$\begin{aligned}
\sum_{i=1}^{(n+1)} i^2 &= \frac{(n+1) \cdot ((n+1)+1) \cdot (2 \cdot (n+1)+1)}{6} \\
\sum_{i=1}^{n} i^2 + (n+1)^2 &= \frac{(n+1) \cdot (n+2) \cdot (2 \cdot (n+2))}{6} \\
\sum_{i=1}^{n} i^2 + (n+1)^2 &= \frac{(n+1) \cdot (n+2) \cdot (2 \cdot (n+2))}{6}
\end{aligned}$$

von IV)

$$\frac{n \cdot (n+1) \cdot (2n+1)}{6} + (n+1)^2 = \frac{(n+1) \cdot (n+2) \cdot (2 \cdot (n+2))}{6}$$
$$\frac{(n^2+n) \cdot (2n+1) \cdot 6}{6} + (n+1)^2 \cdot 6 = \frac{(n+1) \cdot (n+2) \cdot (2 \cdot (n+2)) \cdot 6}{6}$$
$$6(n+1)^2 + n \cdot (n+1) \cdot (2n+1) = (n+1)(n+2(2(n++1)+1))$$
$$6n^2 + 12n + 6 + 2n^3 + 3n^2 + n = (n^2+3n+2) \cdot (2n+3)$$
$$2n^3 + 4n^2 + 13n + 6 = 2n^3 + 9n^2 + 13n + 6$$

b)

$$LENGTH(OVERWRITE(a, L)) = LENGTH(L)$$

I-IA)

$$LENGTH(OVERWRITE(a, L)) = LENGTH(CREATE())$$
$$0 = 0$$

II-IV)

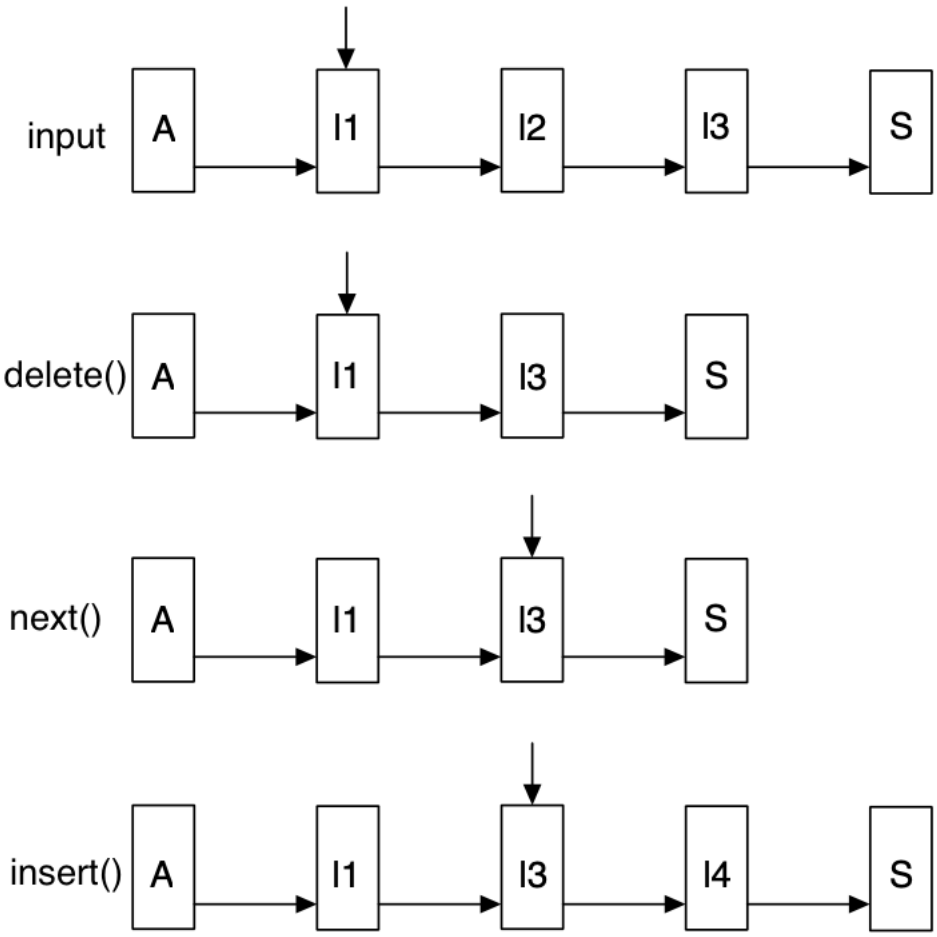
$$LENGTH(OVERWRITE(a, L)) = LENGTH(L)$$

III-IS)

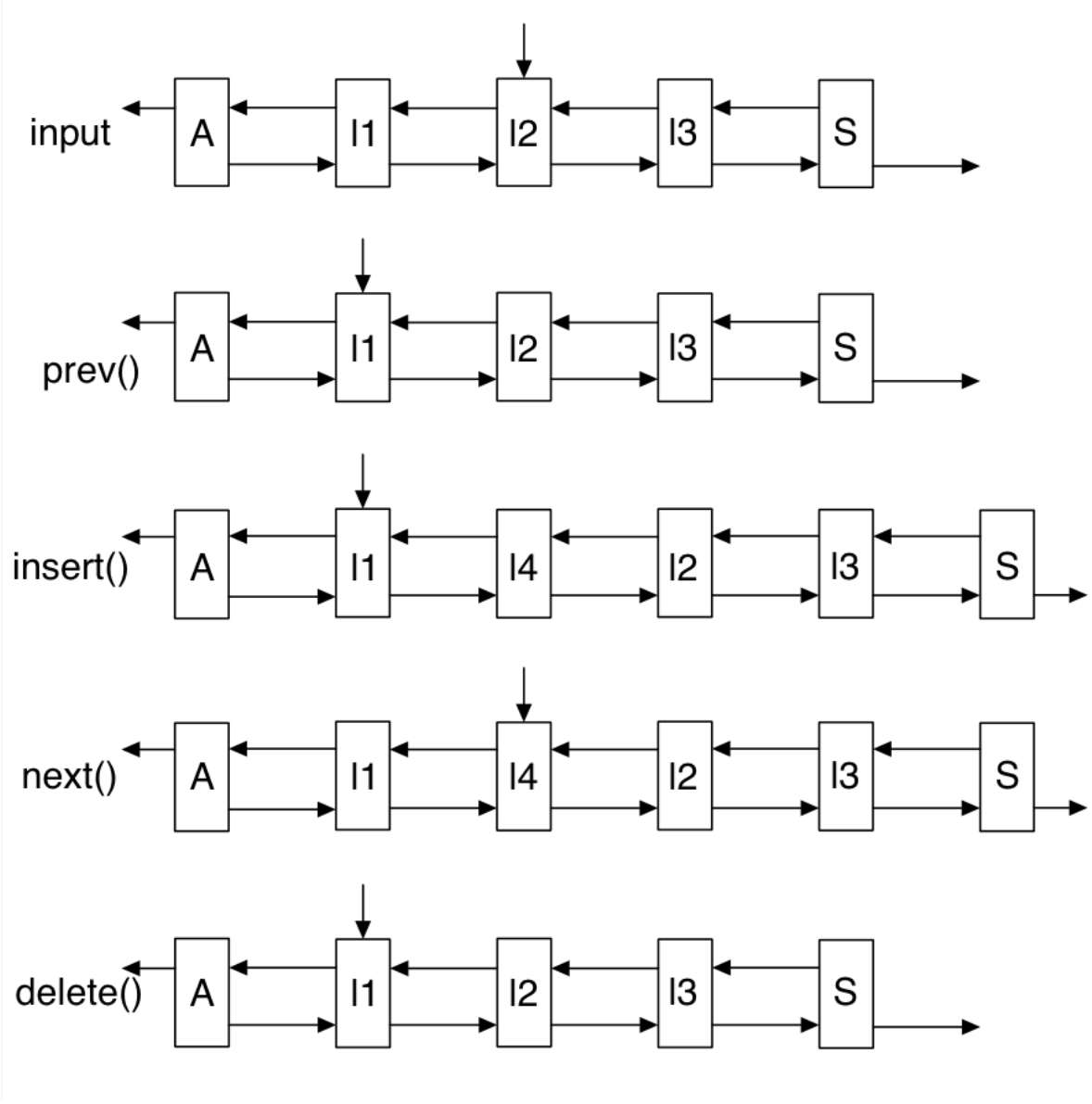
$$z.z.\ LENGTH(OVERWRITE(a, INSERT(b, L)) = LENGTH(INSERT(b, L))$$
$$LENGTH(INSERT(a, L)) = 1 + LENGTH(L)$$
$$1 + LENGTH(L) = 1 + LENGTH(L)$$

Aufgabe 4

a)



b)


input
A
I1
I2
I3
S
prev()
A
I1
I2
I3
S
insert()
A
I1
I4
I2
I3
S
next()
A
I1
I4
I2
I3
S
delete()
A
I1
I2
I3
S

Aufgabe 5

blatt01-python

April 20, 2022

1 Datenstrukturen und Algorithmen

1.1 Praktische Aufgabe 1

In dieser praktischen Aufgabe werden Sie lineare Strukturen (einfach und doppelt verkettete Listen) implementieren. Diese Aufgabe dient dazu einige Konzepte der Vorlesung zu wiederholen.

Die Abgaben werden mit der `nbgrader` Erweiterung korrigiert. Das System erwartet, dass der Code zum Lösen der Aufgaben nach der `#YOUR CODE HERE` Anweisung kommt. Außerdem darf die Zellenreihenfolge nicht geändert werden. Damit Sie selbst Ihre Lösungsvorschläge validieren können, werden Ihnen Unittests zur Verfügung gestellt. Beachten Sie, dass diese Tests keine Garantie sind für das Erreichen der vollen Punktzahl, da Sie nur einen Teil der Funktionalität prüfen.

Wichtig: Füllen Sie auch die erste Zelle mit dem Titel Abgabeteam vollständig aus. Dies ermöglicht uns auch bei technischen Problemen die Abgaben eindeutig zuordnen zu können.

Übersicht der Aufgaben (20 Punkte):

1. **Einfach verkettete Listen** - insgesamt: 2 Punkte
    1. insert() - 1P.
    2. delete() - 1P.
2. **Doppelt verkettete Listen** - insgesamt: 18 Punkte
    1. insert() - 1P.
    2. delete() - 1P.
    3. length() - 2P.
    4. join() - 2P.
    5. split() - 3P.
    6. cycle() - 3P.
    7. swap() - 2P.
    8. reverse() - 4P.

1.2 Abgabeteam

Tragen Sie in die unterstehende Zelle Ihre Daten ein:

23

Mohammed Al-Laktah + 419664

Salah Atallah + 414867

1.3 Module importieren

Zuerst werden die benötigten Module importiert. Sie dürfen keine weiteren Module importieren.

Wenn in Ihrer Entwickungsumbegung (z.B Google Colab oder Deepnote) bestimmte Module nicht verfügbar sind, dann kommentieren Sie die erste Zeile aus um die Module in der Laufzeitumgebung temporär zu installieren.

YOUR ANSWER HERE

[28]:
```
#!pip install nose
from nose.tools import assert_equal
```

2 Einfach verkettete Listen

Der folgende Code implementiert teilweise einfach verkettete Listen. Ihre Aufgabe ist es die fehlenden Funktionen *insert()* und *delete()* zu implementieren, so wie sie in der Vorlesung vorgestellt worden sind.

Der zur Verfügung gestellte Code definiert zunächst die Klasse *SinglyLinkedList* und *Node*, sowie die Funktionen *create()*, *get()*, *reset()*, *next()*, *isEmpty()* und *isLast()*, die bereits aus der Vorlesung bekannt sind.

Desweiteren verwenden wir die folgende Darstellung für einfach verkettete Listen und deren Elemente: Die Pointer auf das nächste Element sind durch `-->` dargestellt. Das 'Anchor' Objekt wird durch `A` representiert und der Marker auf das aktuelle Element der Liste wird mit `( )` dargestellt.

```
Node:               SinglyLinkedList - create():
   | |-->              | |-->
   |?|                 (A)
```

[29]:
```
class Node:
    """ A single object which holds an item and a pointer to its successor. """
    def __init__(self, item):
        self.item = item
        self.next = None

    def __repr__(self):
        return str(self.item)


class SinglyLinkedList:
    """ A list object which points to the current node element and the head of␣
 ↪the list. """
    def __init__(self):
        self.head = Node("Anchor")
        self.cur = self.head

    def __repr__(self):
```

```
        str_out = str(self.head) if self.head != self.cur else f"({str(self.
↪head)})"
        x = self.head.next
        while x:
            str_out += f", {str(x)}" if self.cur != x else f", ({str(x)})"
            x = x.next
        return str_out


def create():
    """ Creates an empty list. """
    return SinglyLinkedList()


def get(list):
    """ Returns the object, the pointer is assigned to."""
    return list.cur.item


def reset(list):
    """ Resets the pointer at the head of the list. """
    list.cur = list.head
    return list


def next(list):
    """ Moves the pointer to the next list item. """
    list.cur = list.cur.next
    return list


def isEmpty(list):
    """ Returns `True` if the list is empty. """
    return list.head.next is None


def isLast(list):
    """ Returns `True` if the pointer points to the last element in the list. """
    return list.cur.next is None
```

Sie können die Funktion *print()* verwenden, um eine List auszugeben.

[30]:
```
my_list_variable = create()
print(my_list_variable)
```

(Anchor)

Bitte implementieren Sie die Funktionen *insert()* und *delete()* in den unten stehenden Zellen an den markierten Stellen.

2.1 a) *insert()* - 1P.

Die Funktion *insert()* soll ein neues *Node* Element erzeugen. Das neue *Node* Element soll hinter dem Marker, wie in der Vorlesung vorgestellt, eingefügt werden. Am Ende wird die aktualisierte Liste zurückgegeben.

Bitte verwenden Sie die unten stehenden Unittests, um Ihren Code zu validieren. Die zur Verfügung gestellten Unittests überprüfen immer nur einen Teil der geforderten Funktionalität und garantieren nicht, dass Sie auch die volle Punktzahl erhalten.

Die folgende Abbildung zeigt das Verhalten der Funktion *insert()* an einem Beispiel:

```
Einfügen von Element '2'

Eingabe:                                  Ausgabe:
   | |-->| |-->| |-->| |-->              | |-->| |-->| |-->| |-->| |-->
   |A|   (1)   |3|   |4|                 |A|   (1)   |2|   |3|   |4|
```

[31]:
```
def insert(item, list):
    """ Inserts the provided item at the current pointer position of the list␣
 ↪element. """
    # YOUR CODE HERE
    node = Node(item)
    node.next = list.cur.next
    list.cur.next = node

    # raise NotImplementedError()
    return list
```

[32]:
```
list = create()
assert_equal(str(list), "(Anchor)")

list = insert(1, insert(2, insert(3, list)))
assert_equal(str(list), "(Anchor), 1, 2, 3")
```

2.2 b) *delete()* - 1P.

Die Funktion *delete()* löscht das Element nach dem Marker und gibt die modifizierte Liste zurück. Sie müssen das Element selbst nicht explizit löschen. Sobald es nicht mehr erreichbar ist, wird es automatisch von Python aus dem Speicher gelöscht, da Python über Garbage Collection verfügt. Bitte implementieren Sie die Funktion so, wie in der Vorlesung vorgestellt.

Die folgende Abbildung zeigt das Verhalten der Funktion *delete()* an einem Beispiel:

```
Löschen eines Elements

Eingabe:                                  Ausgabe:
   | |-->| |-->| |-->| |-->| |-->        | |-->| |-->| |-->| |-->
   |A|   (1)   |2|   |3|   |4|           |A|   (1)   |3|   |4|
```

[33]:
```
def delete(list):
    """ Deletes the item  """
    # YOUR CODE HERE
    if list.cur.next:
        list.cur.next = list.cur.next.next
    # raise NotImplementedError()
    return list
```

[34]:
```
list = insert(1, insert(2, insert(3, insert(4, create()))))

list = delete(next(list))
assert_equal(str(list), "Anchor, (1), 3, 4")
```

3 Doppelt verkettete Listen

Der folgende Code implementiert eine doppelt verkettete Liste mit einem 'Anchor' und 'Sentinel' Element. Ihre Aufgabe ist es die fehlenden Funktionen *insert()* und *delete()* zu implementieren, so wie sie in der Vorlesung vorgestellt worden sind. Zusätzlich sollen Sie die folgenden Funktionen implementieren, die später noch näher erklärt werden: *length()*, *join()*, *split()*, *cycle()*, *swap()* und *reverse()*.

Der zur Verfügung gestellte Code definiert zunächst die Klassen *DoublyLinkedList*, *Node* sowie die Funktionen *create()*, *get()*, *reset()*, *next()*, *previous()*, *isEmpty()* und *isLast()*, die bereits aus der Vorlesung bekannt sind.

Desweiteren verwenden wir die folgende Darstellung für doppelt verkettete Listen und deren Elemente: Die Pointer auf das nächste und vorherige Element werden jeweils durch `-->` und `<--` dargestellt. Das 'Anchor' und 'Sentinel' Objekt werden jeweils durch `A` und `S` representiert und der Marker auf das aktuelle Element der Liste wird mit `( )` dargestellt.

```
Node:                DoublyLinkedList - create():
   | |-->                | |-->| |-->
   |?|                    (A)   |S|
<--| |                 <--| |<--| |
```

[35]:
```
class Node:
    def __init__(self, item):
        self.item = item
        self.prev = None
        self.next = None

    def __repr__(self):
        return str(self.item)


class DoublyLinkedList:
    def __init__(self):
        self.head = Node("Anchor")
```

```
        self.tail = Node("Sentinel")
        self.head.next = self.tail
        self.tail.prev = self.head
        self.cur = self.head

    def __repr__(self):
        str_out = str(self.head) if self.head != self.cur else f"({str(self.
↪head)})"
        x = self.head.next
        while x:
            str_out += f", {str(x)}" if self.cur != x else f", ({str(x)})"
            x = x.next
        return str_out

def create():
    return DoublyLinkedList()


def get(list):
    return list.cur.item


def reset(list):
    list.cur = list.head
    return list


def next(list):
    list.cur = list.cur.next
    return list


def prev(list):
    list.cur = list.cur.prev
    return list


def isEmpty(list):
    return list.head.next.item == "Sentinel"


def isLast(list):
    return list.cur.next.item == "Sentinel"
```

3.1 a) *insert()* - 1P.

Die Funktion *insert()* soll ein neues *Node* Element erzeugen. Das neue *Node* Element soll hinter dem Marker, wie in der Vorlesung vorgestellt, eingefügt werden. Am Ende wird die aktualisierte Liste zurückgegeben.

Beachten Sie, dass wenn der Marker auf das "Sentinel"-Element zeigt, die Funktion *insert()* nichts tun soll. Typischerweise würde man in so einer Situation einen Fehler ausgeben. Der Einfachheit halber verzichten wir an dieser Stelle darauf.

Die folgende Abbildung zeigt das Verhalten der Funktion *insert()* an einem Beispiel:

```
Einfügen von Element '2'

Eingabe:                                    Ausgabe:
   | |-->| |-->| |-->| |-->| |-->             | |-->| |-->| |-->| |-->| |-->| |-->
   |A|   (1)   |3|   |4|   |S|               |A|   (1)   |2|   |3|   |4|   |S|
<--| |<--| |<--| |<--| |<--| |            <--| |<--| |<--| |<--| |<--| |<--| |
```

[36]:
```
def insert(item, list):
    # YOUR CODE HERE
    if list.cur == list.tail:
        return list

    node = Node(item)
    node.prev = list.cur
    node.next = list.cur.next
    node.prev.next = node
    node.next.prev = node

    # raise NotImplementedError()
    return list
```

[37]:
```
list = insert(4, insert(3, insert(2, insert(1, create()))))
assert_equal(str(list), "(Anchor), 4, 3, 2, 1, Sentinel")
```

3.2 b) *delete()* - 1P.

Die Funktion *delete()* löscht das aktuelle Element auf das der Marker zeigt und gibt die modifizierte Liste zurück. Sie müssen das Element selbst nicht explizit löschen. Sobald es nicht mehr erreichbar ist, wird es automatisch von Python aus dem Speicher gelöscht, da Python über Garbage Collection verfügt. Bitte implementieren Sie die Funktion so, wie in der Vorlesung vorgestellt.

Bitte beachten Sie, dass sowohl das Anchor als auch das Sentinel Element nicht gelöscht werden dürfen. Falls dies nach der obigen Spezifikation passieren würde, dann tut die Funktion *delete()* nichts. Typischerweise würde man in so einer Situation einen Fehler ausgeben. Der Einfachheit halber verzichten wir an dieser Stelle darauf.

Die folgende Abbildung zeigt das Verhalten der Funktion *delete()* an einem Beispiel:

```
Löschen eines Elements
```

```
Eingabe:                                          Ausgabe:
   | |-->| |-->| |-->| |-->| |-->| |-->             | |-->| |-->| |-->| |-->| |-->
   |A|   |1|   (2)   |3|   |4|   |S|               |A|   (1)   |3|   |4|   |S|
<--| |<--| |<--| |<--| |<--| |<--| |            <--| |<--| |<--| |<--| |<--| |
```

[38]:
```
def delete(list):
    # YOUR CODE HERE
    if (list.cur in [list.head, list.tail]):
        return list
    list.cur.prev.next = list.cur.next
    list.cur.next.prev = list.cur.prev
    prev(list)
    # raise NotImplementedError()
    return list
```

[39]:
```
list = insert(4, insert(3, insert(2, insert(1, create()))))
list = delete(next(list))
assert_equal(str(list), "(Anchor), 3, 2, 1, Sentinel")

list = create()
list = delete(list)
assert_equal(str(list), "(Anchor), Sentinel")
```

3.3 c) *length()* - 2P.

Die Funktion *length()* soll die aktuelle Länge einer Liste zurückgeben. Es ist zu beachten, dass das 'Anchor' und 'Sentinel' Objekt selbst nicht zu der Anzahl der Elemente einer List zählen sollen. Eine leere Liste soll also die Länge 0 haben. Wenn einer Liste 4 Elemente zugewiesen worden sind, dann soll die Länge dieser Liste als 4 angegeben werden.

Weiter ist zu beachten, dass die Funktion weder den Marker der List verändern darf noch die Liste auf eine andere Art modifizieren darf.

Die folgende Abbildung zeigt das Verhalten der Funktion *length()* anhand von Beispielen:

```
Liste 1:                          Liste 2:
   | |-->| |-->                      | |-->| |-->| |-->| |-->| |-->| |-->
   (A)   |S|                         |A|   |1|   (2)   |3|   |4|   |S|
<--| |<--| |                      <--| |<--| |<--| |<--| |<--| |<--| |

Length: 0                         Length: 4
```

[40]:
```
def length(list):
    # YOUR CODE HERE
    length = 0
    marker = list.cur
    reset(list)
    while(not isLast(list) and not isEmpty(list)):
```

```
        length += 1
        next(list)
    list.cur = marker
    # raise NotImplementedError()
    return length
```

[41]:
```
list = create()
assert_equal(length(list), 0)

list = insert(4, insert(3, insert(2, insert(1, create()))))
assert_equal(length(list), 4)
```

3.4 d) *join()* - 2P.

Die Funktion *join()* bekommt zwei Listen übergeben und gibt die Verschmelzung beider Listen zurück. Hierbei soll die zweite Liste an das Ende der ersten Liste angehängt werden und die erste Liste zurückgegeben werden. Bitte beachten Sie, dass sie zusätzlich sicher stellen müssen, dass die 'Anchor' und 'Sentinel' Objekte in der List korrekt sind. Außerdem müssen die entsprechenden Marker/Pointer auf diese Objekte konsistent sein.

Die folgende Abbildung zeigt das Verhalten der Funktion *join()* an einem Beispiel:

```
Liste 1:                                Liste 2:
   | |-->| |-->| |-->| |-->| |-->          | |-->| |-->| |-->| |-->| |-->
   |A|   (1)   |2|   |3|   |S|             |A|   |4|   (5)   |6|   |S|
<--| |<--| |<--| |<--| |<--| |          <--| |<--| |<--| |<--| |<--| |

Ausgabe:
   | |-->| |-->| |-->| |-->| |-->| |-->| |-->| |-->
   |A|   (1)   |2|   |3|   |4|   |5|   |6|   |S|
<--| |<--| |<--| |<--| |<--| |<--| |<--| |<--| |
```

[42]:
```
def join(list, list2):
    """ Joins two lists into a single one. """
    # YOUR CODE HERE
    list.tail.prev.next = list2.head.next
    list2.head.next.prev = list.tail.prev
    list.tail = list2.tail
    # raise NotImplementedError()
    return list
```

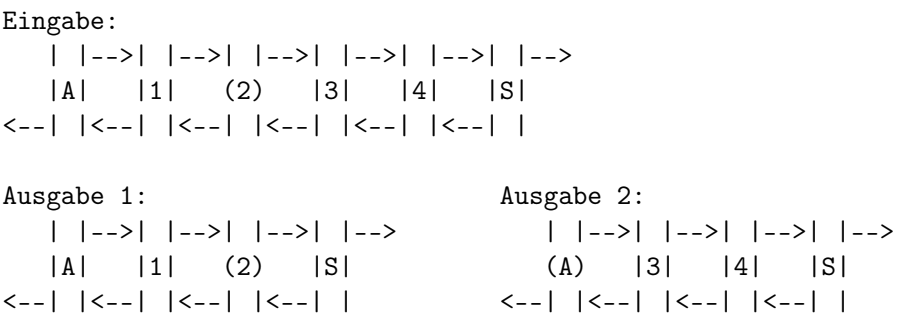
[43]:
```
list1 = insert(1, insert(2, insert(3, insert(4, create()))))
list2 = insert(5, insert(6, insert(7, insert(8, create()))))

list = join(list1, list2)
assert_equal(str(list), "(Anchor), 1, 2, 3, 4, 5, 6, 7, 8, Sentinel")
```

3.5 e) *split()* - 3P.

Die Funktion *split()* soll eine Liste als Eingabe erhalten und zwei Listen zurückgeben. Hierbei soll die übergegebene Liste an der Stelle des Markers in zwei Listen aufgeteilt werden. Bitte implementieren Sie die folgende Funktion, indem Sie eine neue Liste erstellen und die Pointer entsprechend umbiegen. Alle Pointer, inklusive der Pointer auf den Anfang und das Ende beider Listen, müssen nach der Ausgabe konsistent sein.

Bitte beachten Sie alle Sonderfälle, wie das Teilen am Anfang oder am Ende der Liste, oder das Teilen einer leeren Liste.

Die folgende Abbildung zeigt das Verhalten der Funktion *split()* an einem Beispiel:



[44]:
```python
def split(list):
    """ Splits the list into two list at the position of the current pointer. """
    # YOUR CODE HERE
    list2 = create()

    if list.cur.next == list.tail or list.cur == list.tail:
        return list, list2

    if list.cur == list.head:
        return list2, list

    start = list.cur.next
    end = list.tail.prev
    list.cur.next = list.tail
    list.tail.prev=list.cur
    list2.head.next = start
    start.prev = list2.head
    list2.tail.prev = end
    end.next = list2.tail

    # raise NotImplementedError()
    return list, list2
```

[45]:
```python
list = insert(1, insert(2, insert(3, insert(4, create()))))
list1, list2 = split(list)
assert_equal(str(list1), "(Anchor), Sentinel")
```

```
assert_equal(str(list2), "(Anchor), 1, 2, 3, 4, Sentinel")

list = insert(1, insert(2, insert(3, insert(4, create()))))
list = next(next(list))
list1, list2 = split(list)
assert_equal(str(list1), "Anchor, 1, (2), Sentinel")
assert_equal(str(list2), "(Anchor), 3, 4, Sentinel")

list = create()
list1, list2 = split(list)
assert_equal(str(list1), "(Anchor), Sentinel")
assert_equal(str(list2), "(Anchor), Sentinel")

list = next(create())
list1, list2 = split(list)
assert_equal(str(list1), "Anchor, (Sentinel)")
assert_equal(str(list2), "(Anchor), Sentinel")
```

3.6 f) *cycle()* - 3P.

Die Funktion *cycle()* soll die Elemente der Liste um eine Stelle nach vorne rotieren. Dies ist eine globale Operation, die den Marker auf das momentane Objekt nicht verändern sollte. Zudem sollten sich am Ende der Operation sowohl das 'Anchor', als auch 'Sentinel' Element an den Enden der Liste befinden. Bitte implementieren Sie diese Funktion gemäß der Beschreibung und der unten stehenden Abbildungen.

Die folgende Abbildung zeigt das Verhalten der Funktion *cycle()* an einem Beispiel:

```
Eingabe:
   | |-->| |-->| |-->| |-->| |-->| |-->| |-->| |-->
   |A|   (1)   |2|   |3|   |4|   |5|   |6|   |S|
<--| |<--| |<--| |<--| |<--| |<--| |<--| |<--| |

Ausgabe:
   | |-->| |-->| |-->| |-->| |-->| |-->| |-->| |-->
   |A|   |6|   (1)   |2|   |3|   |4|   |5|   |S|
<--| |<--| |<--| |<--| |<--| |<--| |<--| |<--| |
```

```
Eingabe:
   | |-->| |-->
   (A)   |S|
<--| |<--| |

Ausgabe:
   | |-->| |-->
   (A)   |S|
<--| |<--| |
```

[46]:
```
def cycle(list):
    """ Cycles the list by one element forward, such that the last element in␣
 ↪the list becomes the first one. """
    # YOUR CODE HERE
    if isEmpty(list):
        return list

    first = list.head.next
    last  = list.tail.prev
```

```
    last.prev.next = list.tail

    list.head.next = last
    list.tail.prev = last.prev

    first.prev = last

    last.next = first
    last.prev = list.head

    # raise NotImplementedError()
    return list
```

[47]:
```
list = insert(1, insert(2, insert(3, insert(4, create()))))
list = cycle(list)
assert_equal(str(list), "(Anchor), 4, 1, 2, 3, Sentinel")
```

3.7 g) *swap()* - 2P.

Die Funktion *swap()* nimmt als Eingabe eine Liste und vertauscht das Element, auf das der Marker zeigt mit dem nächsten Element und gibt die neue Liste zurück. Bitte implementieren Sie die Funktion *swap()* gemäß der Beschreibung und der unten stehenden Abbildung. Beachten Sie zudem alle Sonderfälle, die auftreten könnten.

Hierbei ist zu beachten, dass die 'Anchor' und 'Sentinel' Elemente nicht vertauscht werden dürfen. Das bedeutet bei leeren Listen oder der Liste mit einem Element tut die Funktion *swap()* nichts.

Die folgende Abbildung zeigt das Verhalten der Funktion *swap()* anhand von Beispielen:

```
Eingabe:                                    Eingabe:
   | |-->| |-->| |-->| |-->| |-->             | |-->| |-->| |
   |A|   (1)   |2|   |3|   |S|               |A|   (1)   |S|
<--| |<--| |<--| |<--| |<--| |            <--| |<--| |<--| |

Ausgabe:                                    Ausgabe:
   | |-->| |-->| |-->| |-->| |-->             | |-->| |-->| |-->
   |A|   |2|   (1)   |3|   |S|               |A|   (1)   |S|
<--| |<--| |<--| |<--| |<--| |            <--| |<--| |<--| |
```

[48]:
```
def swap(list):
    """ Swap the current element with the next element in the list. """
    # YOUR CODE HERE

    if list.cur in [list.head, list.tail] or list.cur.next == list.tail:
        return list

    curprev= list.cur.prev          #head
    cur=list.cur                    #1
```

```
    curnext=list.cur.next           #2
    curnextnext=list.cur.next.next  #3

    cur.next = curnextnext          #RED
    cur.prev = curnext              #Blue

    curnext.prev = curprev          #Yellow
    curnext.next = cur              #Dark Green

    curprev.next = curnext          #Orange

    curnextnext.prev = cur          #Light Green

    # raise NotImplementedError()
    return list
```

[49]:
```
list = next(insert(1, insert(2, insert(3, insert(4, create())))))
list = swap(list)
assert_equal(str(list), "Anchor, 2, (1), 3, 4, Sentinel")
```

3.8 h) *reverse()* - 4P.

Die Funktion *reverse()* bekommt eine Liste als Eingabe und gibt eine Liste zurück, wobei die Reihenfolge der Elemente umgekehrt wirde. Hierbei soll der Marker auf das aktuelle Element erhalten bleiben.

Es ist zu beachten, dass die 'Anchor' und 'Sentinel' Elemente ihre Position beibehalten sollen. Das bedeutet bei leeren Listen oder der Liste mit einem Element tut die Funktion *reverse()* nichts.

Hilfestellung: Wir empfehlen die Aufgabe mit Hilfe einer Schleife zu lösen, die über alle Elemente der Liste iteriert und die entsprechenden Zeiger der Liste umbiegt. Das erste und letzte Element der Liste sind besonders zu behandeln.

Die folgende Abbildung zeigt das Verhalten der Funktion *reverse()* anhand von Beispielen:

```
Eingabe:                                  Eingabe:
   | |-->| |-->| |-->| |-->| |-->           | |-->| |-->| |
   |A|   (1)   |2|   |3|   |S|             (A)   |1|   |S|
<--| |<--| |<--| |<--| |<--| |           <--| |<--| |<--| |

Ausgabe:                                  Ausgabe:
   | |-->| |-->| |-->| |-->| |-->           | |-->| |-->| |-->
   |A|   |3|   |2|   (1)   |S|             (A)   |1|   |S|
<--| |<--| |<--| |<--| |<--| |           <--| |<--| |<--| |
```

[50]:
```
def reverse(list):
    """ Reverse the order of elements in the list. """
    # YOUR CODE HERE
    if isEmpty(list):
```

```
        return list

    temp = None
    current = list.head
    list.head = list.tail
    list.tail = current
    list.head.item = 'Anchor'
    list.tail.item = 'Sentinel'
    if list.cur == list.head:
        list.cur = list.tail
    if list.cur == list.tail:
        list.cur = list.head

    while current is not None:
        temp = current.prev
        current.prev = current.next
        current.next = temp
        current = current.prev
    # raise NotImplementedError()
    return list
```

[51]:
```
list = insert(1, insert(2, insert(3, insert(4, create()))))
list = reverse(list)
assert_equal(str(list), "(Anchor), 4, 3, 2, 1, Sentinel")

i = "a"
list = insert(i, insert(2, insert(i, insert(4, create()))))
list = next(next(next(list)))
list = reverse(list)
assert_equal(str(list), "Anchor, 4, (a), 2, a, Sentinel")

list = create()
list = reverse(list)
assert_equal(str(list), "(Anchor), Sentinel")

list = create()
next(list)
list = reverse(list)
assert_equal(str(list), "Anchor, (Sentinel)")

list = (insert (1, create()))
next(list)
list = reverse(list)
assert_equal(str(list), "Anchor, (1), Sentinel")
```

3.9 Jupyter Notebook Stolperfalle

Bei der Benutzung von Jupyter Notebooks, wird der globale Zustand aller Variablen zwischen der Ausführung von verschiedenen Zellen erhalten. Dies ist auch der Fall, wenn Zellen gelöscht oder hinzugefügt werden. Um sicher zu gehen, dass nicht ausversehen notwendige Variablen überschrieben oder gelöscht wurden, kann der Befehl `Kernel -> Restart & Run All` ausgeführt werden.