Musterlösung zur Übung 3

Aufgabe 1

a) Für die Tabelle ergibt sich:

x	y	q	r	r+y*q
31	9	0	31	31+9*0=31
31	9	1	22	22+9*1=31
31	9	2	13	13+9*2=31
31	9	3	4	4+9*3=31

Aufgabe 2

a) Die Implementation für archimedes1(k) startet bei einem Quadrat und führt danach k Verdopplungen durch. Für jede Verdopplung wird die Eckenzahl n, der untere Schätzwert s und der obere Schätzwert t sowie die Differenz ausgegeben:

```
def archimedes1(k):
    s,t,n = sqrt(2),2,4 #Startwerte: Quadrat
    print 'n:',n,' s', n/2*s,' < pi < t: ',n/2*t,' Dif: ',n/2*t-n/2*s
    for i in range(0,k):
        n *= 2
        s = sqrt(2.0 - sqrt(4.0 - s**2))
        t = 2.0 / t * (sqrt(4.0 + t**2) - 2.0)
        print 'n:',n,' s', n/2*s,' < pi < t: ',n/2*t,' Dif: ',n/2*t-n/2*s
```

b) Die Ausgabe von archimedes1(30) sieht folgendermaßen aus:

```
n: 4  s: 2.82842712475 < pi < t: 4 Dif: 1.17157287525
n: 8  s: 3.06146745892 < pi < t: 3.31370849898 Dif: 0.252241040064
n: 16  s: 3.12144515226 < pi < t: 3.18259787807 Dif: 0.0611527258165
n: 32  s: 3.13654849055 < pi < t: 3.15172490743 Dif: 0.0151764168833
n: 64  s: 3.14033115695 < pi < t: 3.14411838525 Dif: 0.00378722829113
n: 128  s: 3.14127725093 < pi < t: 3.14222362994 Dif: 0.000946379009588
n: 256  s: 3.14151380114 < pi < t: 3.14175036917 Dif: 0.000236568025558
n: 512  s: 3.14157294037 < pi < t: 3.1416320807 Dif: 5.91403343662e-05
n: 1024  s: 3.14158772528 < pi < t: 3.14160251024 Dif: 1.47849620111e-05
n: 2048  s: 3.1415914215 < pi < t: 3.14159511772 Dif: 3.6962137302e-06
n: 4096  s: 3.14159234561 < pi < t: 3.14159326963 Dif: 9.24020668425e-07
n: 8192  s: 3.14159257655 < pi < t: 3.14159280838 Dif: 2.31834680342e-07
n: 16384  s: 3.14159263346 < pi < t: 3.14159269096 Dif: 5.74997205405e-08
n: 32768  s: 3.14159265481 < pi < t: 3.14159267557 Dif: 2.07628638726e-08
n: 65536  s: 3.14159264532 < pi < t: 3.14159269096 Dif: 4.56417530437e-08
n: 131072  s: 3.14159260738 < pi < t: 3.14159252379 Dif: -8.35872460136e-08
n: 262144  s: 3.14159291094 < pi < t: 3.14159086958 Dif: -2.04136055881e-06
n: 524288  s: 3.1415941252 < pi < t: 3.14159252379 Dif: -1.60140671746e-06
n: 1048576  s: 3.1415965537 < pi < t: 3.14160058363 Dif: 4.02992151516e-06
n: 2097152  s: 3.1415965537 < pi < t: 3.14159252379 Dif: -4.02991634596e-06
n: 4194304  s: 3.14167426502 < pi < t: 3.14144515887 Dif: -0.000229106150957
n: 8388608  s: 3.14182968189 < pi < t: 3.1409707956 Dif: -0.000858886286714
n: 16777216  s: 3.14245127249 < pi < t: 3.13398329389 Dif: -0.00846797860878
n: 33554432  s: 3.14245127249 < pi < t: 3.11105678803 Dif: -0.0313944844688
n: 67108864  s: 3.16227766017 < pi < t: 3.05362474789 Dif: -0.10865291228
n: 134217728  s: 3.16227766017 < pi < t: 2.61983729518 Dif: -0.542440364989
n: 268435456  s: 3.46410161514 < pi < t: 3.05362474789 Dif: -0.410476867249
n: 536870912  s: 4.0 < pi < t: 0.0 Dif: -4.0
```

```
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    archimedes1(30)
  File "archimedes.py", line 11, in archimedes1
    t = 2/t*(sqrt(4+t**2)-2)
ZeroDivisionError: float division by zero
```

Beobachtung: Bis zu einer Eckenzahl von n = 32768 sehen die Berechnungen sehr vielversprechend aus. Jedoch tritt bei n = 65536 die erste Unregelmäßigkeit auf. Dort wird der untere Schätzwert wieder kleiner, obwohl es sich rein mathematisch um eine monoton steigende Folge handelt. Nach der nächsten Verdopplung, bei n = 131072 wird das erste Mal die Differenz negativ. Der obere Schätzwert ist nun also niedriger als der untere. Nach einigen weiteren Verdopplungen werden die numerischen Ungenauigkeiten so groß, dass schließlich die obere Schranke auf 0.0 springt. Eine Division durch Null ist die Folge und führt zum Abbruch der Funktion.

Erklärung: Der Grund für dieses merkwürdige Verhalten ist ein Effekt der Fehlerfortpflanzung, die sogenannte nummerische Auslöschung. Diese tritt bei der Subtraktion fast identischer Zahlen auf. Für unseren konkreten Fall konvergieren die Folgen für die innere und äußere Kantenlänge beide gegen Null (je mehr Kanten ein n-Eck hat, desto kürzer ist jede einzelne Kante). Die beiden Terme $\sqrt{4-s_n^2}$ und $\sqrt{4+t_n^2}$ unterscheiden sich folglich mit steigendem n immer weniger von 2. Auf Grund der begrenzten Genauigkeit der Zahlendarstellung im Rechner, wird dieser geringe, aber für die weitere Berechnung entscheidende, Unterschied von Rundungsfehlern beeinflusst. Bei der anschließenden Subtraktion von bzw. mit 2 stimmen die höherwertigen Stellen überein und es löschen sich viele der gültigen Stellen aus. Das Ergebnis wird nun ausschließlich durch die Rundungsfehler bestimmt.

c) Zunächst zeigen wir, dass der Satz an Formeln äquivalent zum ersten ist. Dafür beginnen wir mit den in Aufgabenteil a) gegebenen Formeln und erweitern geschickt mit Eins, so dass wir die dritte Binomische Formel anwenden können:

$$s_{2n} = \sqrt{2-\sqrt{4-s_n^2}} = \sqrt{2-\sqrt{4-s_n^2}}\frac{\sqrt{2+\sqrt{4-s_n^2}}}{\sqrt{2+\sqrt{4-s_n^2}}} = \frac{\sqrt{4-(4-s_n^2)}}{\sqrt{2+\sqrt{4-s_n^2}}} = \frac{s_n}{\sqrt{2+\sqrt{4-s_n^2}}}$$

Analoge Rechnung für die obere Schranke:

$$t_{2n} = \frac{2}{t_n}\left(\sqrt{4+t_n^2}-2\right) = \frac{2}{t_n}\left(\sqrt{4+t_n^2}-2\right)\frac{\sqrt{4+t_n^2}+2}{\sqrt{4+t_n^2}+2} = \frac{2}{t_n}\,\frac{(4+t_n^2)-4}{\sqrt{4+t_n^2}+2}$$
$$= \frac{2t_n}{\sqrt{4+t_n^2}+2}$$

Die Implementation archimedes2(k) verwendet den neuen Satz an Formeln:

```
def archimedes2(k):
    s,t,n = sqrt(2),2,4 #Startwerte: Quadrat
    print 'n:',n,' s:', n/2*s,'< pi < t:',n/2*t,'Dif:',n/2*t-n/2*s
    for i in range(0,k):
        n *= 2
        s = s / sqrt(2.0 + sqrt(4.0 - s**2))
        t = (2.0*t) / (sqrt(4.0 + t**2) + 2.0)
        print 'n:',n,' s:', n/2*s,'< pi < t:',n/2*t,'Dif:',n/2*t-n/2*s
```

Diesmal tritt bei einem Testlauf kein unerwartetes Verhalten auf. Der neue Satz an Formeln umgeht das Problem der Auslöschung, indem die Differenz zweier fast identischer Zahlen vermieden wird. Auch wenn die beiden Formeln mathematisch äquivalent sind, verhalten sie

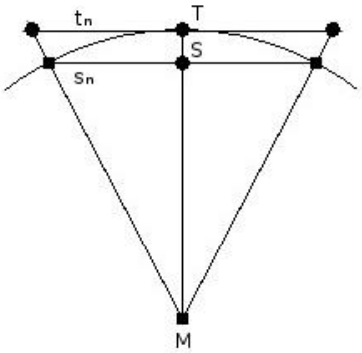
sich numerisch unterschiedlich. Es lohnt sich also folglich, bevor man Formeln implementiert, diese genauer zu untersuchen. Oft kann man durch eine geschickte Wahl der Reihenfolge der einzelnen Rechenschritte oder durch Umformungen das Problem der Auslöschung umgehen.

Betrachten wir die Ausgabe (n: 2097152 s: 3.14159265359 < pi < t: 3.14159265359 Dif: 3.52518014779e-12), so sehen wir dass für n = 2087152, bereits alle 12 ausgegebenen Dezimalstellen der unteren und oberen Schranke übereinstimmen. Wir erhalten somit 12 Dezimalstellen nach 19 Verdopplungen, also etwa zwei weitere gültige Dezimalstellen alle drei Verdopplungen.

d) Die gegebene Formel lässt sich elegant mit dem Strahlensatz berechnen. Dazu betrachten wir folgende Skizze. Wir befinden uns im Einheitskreis, daher gilt $\overline{TM} = r = 1$:

Der Strahlensatz gibt uns:

$\frac{t_n}{s_n} = \frac{\overline{TM}}{\overline{SM}} = \frac{1}{\overline{SM}} \rightarrow t_n = \frac{s_n}{\overline{SM}}$



Für $\overline{SM}$ bemühen wir noch Pythagoras:

$$\overline{SM} = \sqrt{1 - \frac{s_n^2}{4}} = \frac{1}{2}\sqrt{4 - s_n^2}$$

Setzen wir dieses Ergebnis ein, erhalten wir:

$$t_n = \frac{2s_n}{\sqrt{4 - s_n^2}}$$

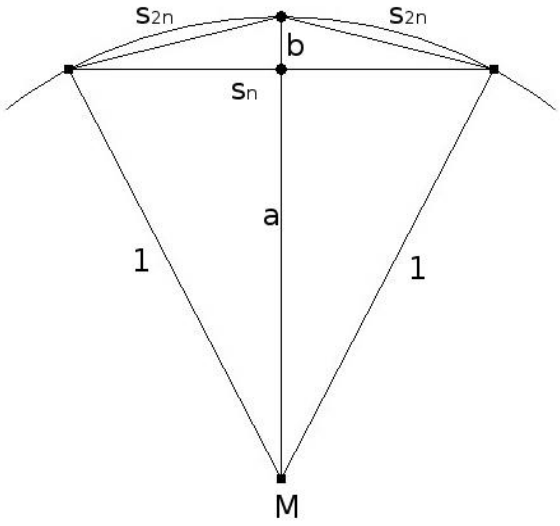
Eine mögliche Implementation für einen Test berechnet mit obiger Formel t_n aus s_n und vergleicht das Ergebnis mit dem direkt berechneten Wert. Dabei ist es wichtig, den Vergleich nicht mit „==“ zu machen, sondern eine gewisse Toleranz zuzulassen. Für einen einfachen Test muss die bisherige Umsetzung nur durch eine if-Abfrage erweitert werden:

```
from math import sqrt

def archimedes1(k):
    s,t,n = sqrt(2),2,4 #Startwerte: Quadrat
    print 'n:',n,' s:', n/2*s,'< pi < t:',n/2*t,'Dif:',n/2*t-n/2*s
    for i in range(0,k):
        if abs(t – 2.0*s / sqrt(4.0 - s**2)) < 1e-15: #Test
            n *= 2
            s = sqrt(2.0 - sqrt(4.0 - s**2))
            t = 2.0 / t * (sqrt(4.0 + t**2) – 2.0)
            print 'n:',n,' s:', n/2*s,'< pi < t:',n/2*t,'Dif:',n/2*t-n/2*s
        else:
           print 'Fehler bei n =', n
           break

def archimedes2(k):
    s,t,n = sqrt(2),2,4 #Startwerte: Quadrat
    print 'n:',n,' s:', n/2*s,'< pi < t:',n/2*t,'Dif:',n/2*t-n/2*s
    for i in range(0,k):
        if abs(t – 2.0*s / sqrt(4.0 - s**2)) < 1e-15: #Test
            n *= 2
            s = s / sqrt(2.0 + sqrt(4.0 - s**2))
            t = (2.0*t) / (sqrt(4.0 + t**2) + 2.0)
            print 'n:',n,' s:', n/2*s,'< pi < t:',n/2*t,'Dif:',n/2*t-n/2*s
        else:
           print 'Fehler bei n =', n
           break
```

Bereits bei n = 128 schlägt dieser Test bei archimedes1(k) Alarm, dagegen wird archimedes2(k) verifiziert.

Bonusaufgabe:

Die Herleitung für die Berechnung der Seitenlängen ist mit elementarer Geometrie (Pythagoras) möglich. Dazu betrachten wir die Seitenlängen eines n-Ecks und eines 2n-Ecks. Für die eingezeichneten Strecken gilt:



$$s_{2n}^2 = b^2 + \frac{s_n^2}{4}$$

$$a^2 = 1 - \frac{s_n^2}{4} \rightarrow a = \sqrt{1 - \frac{s_n^2}{4}}$$

$$b = 1 - a = 1 - \sqrt{1 - \frac{s_n^2}{4}}$$

Setzen wir dies nun ineinander ein, erhalten wir:

$$s_{2n}^2 = \left(1 - \sqrt{1 - \frac{s_n^2}{4}}\right)^2 + \frac{s_n^2}{4} = 1 - 2\sqrt{1 - \frac{s_n^2}{4}} + 1$$

$$= 2 - \sqrt{4 - s_n^2}$$

$$s_{2n} = \sqrt{2 - \sqrt{4 - s_n^2}}$$

Die Herleitung für die obere Schranke ergibt sich analog.

Aufgabe 3

Zur Implementation des unittests nimmt man im Prinzip die Lösung checksorting.py aus der letzten Übung und fügt sie in das Beispiel des im Übungsblatt gegebenen URLs ein. Damit ergibt sich folgender Code:

```
import random
import unittest
from muster_sort import *

class TestSortingFunctions(unittest.TestCase):
  def setUp(self):
    self.data = range(10)
    self.data.append(1) # add two entries that ...
    self.data.append(5) #   ... occur twice
    random.shuffle(self.data) # shuffle the sequence
    self.originalData =copy.deepcopy(self.data); #copy the data for comparison

  def testMergeSort(self):
    mergeSort(self.data)
    self.checkSorting(self.originalData, self.data);

  def testSelectionSort(self):
    selectionSort(self.data)
    self.checkSorting(self.originalData, self.data);

  def testQuickSort(self):
    quickSort(self.data)
    self.checkSorting(self.originalData, self.data);
```

```
    def checkSorting(self, before, after):
      print before, after
      afterCopy = copy.deepcopy(after);
      # lengths of both arrays should be the same
      self.assertEqual(len(before), len(after))

      # all values of original array should still exist
      for x in before:
        afterCopy.remove(x); # raises exception if element not found

      # the values should actually be sorted
      for ii in range(1, len(after)):
        self.assertTrue(after[ii-1] <= after[ii])

if __name__ == '__main__':
    unittest.main()
```

Aufgabe 4

a) Die Idee der Implementation ist, den aktuellen Anfang (Variable `head`) und das aktuelle Ende (Variable `tail`) des gültigen Bereichs des internen Arrays `data` zu speichern, wobei man dieses Array als Ringpuffer behandelt. Wird eine dieser Variablen inkrementiert oder dekrementiert, erfolgt die Berechnung immer Modulo der aktuellen Arraygröße, wie in der Aufgabenstellung beschrieben. Ein Problem besteht darin, wie man unterscheidet, ob die Deque voll oder leer ist – bei einer naiven Implementierung wären Beginn (`head`) und Ende (`tail`) in beiden Fällen identisch. Deshalb reservieren wir Speicher für ein Extra-Element, das nie benutzt werden darf (es gilt also: `capacity = len(data)-1`). Eine leere Deque erkennt man jetzt daran, dass `head` und `tail` identisch sind, während für eine volle Deque `tail = (head - 1) % len(data)` gilt. Die aktuelle Anzahl der Elemente in der Deque wird nicht separat gespeichert, sondern aus dem Anfangs- und End-Index berechnet. Die folgenden Vor- und Nachbedingungen gelten

   a. Konstruktor: Deque(N)
      i. Vor: N ist positiver Integer
      ii. Nach: Speicher mit N+1 Elementen allokiert, size() == 0, capacity() == N
   b. size():
      i. Nach: Anzahl der gespeicherten Elemente zurückgegeben
   c. capacity():
      i. Nach: Rückgabe ist aktuelle Kapazität mit size() <= capacity()
   d. push(x):
      i. Nach: Wert hinten angehängt. size um 1 vergrößert
   e. popFirst():
      i. Vor: mindestens ein Element in der Deque
      ii. Nach: erstes Element zurückgegeben und gelöscht. size um 1 verkleinert
   f. popLast():
      i. Vor: mindestens ein Element in der Deque
      ii. Nach: letztes Element zurückgegeben und gelöscht. size um 1 verkleinert

b) Zu den in der Aufgabenstellung verlangten Funktionen wird noch __str__(self) implementiert. Diese Funktion dient zur Ausgabe der Deque in einen String (nützlich für Beispiele und Debuggen).

```
class Deque:
  """A double-ended queue implementation"""

  def __init__(self, initial_capacity=4):
    if initial_capacity <= 0:
        raise ValueError("Deque():initial_capacity must be positive")
    self.data = [None]*(initial_capacity+1)  # reserve internal memory
    self.head = 0                            # pointer to the first element
    self.tail = 0                            # pointer beyond last element

  def size(self):
    return (self.tail - self.head) % len(self.data)

  def capacity(self):
    return len(self.data)-1

  def push(self, x):
    # check whether capacity is depleted
    if self.size() == self.capacity():
        old_capacity = self.capacity()
        new_capacity = old_capacity*2          # double capacity
        new_data = [None]*(new_capacity + 1)   # reserve new memory
        for k in xrange(old_capacity):         # copy the data
            new_data[k] = self.data[(self.head + k) % len(self.data)]
        self.head = 0                          # reset internal pointers
        self.tail = old_capacity
        self.data = new_data

    # save element at the current tail location
    self.data[self.tail] = x
    # increment tail pointer
    self.tail = (self.tail+1) % len(self.data)

  def popFirst(self):
    if not self.size():
      raise RuntimeError('popFirst(): Deque is empty')
    # get first element
    x = self.data[self.head]
    # increment head (= remove first element)
    self.head = (self.head+1) % len(self.data)
    return x

  def popLast(self):
    if not self.size():
      raise RuntimeError('popLast(): Deque is empty')
    # decrement tail (= remove last element)
    self.tail = (self.tail-1) % len(self.data)
    # get last element
    x = self.data[self.tail]
    return x

  def __str__(self):
    if self.head == self.tail:
      return '[]'
    else:
      res ='['+str(self.data[self.head])
      for i in range(self.head+1, self.head+self.size()):
        res +=', '+str(self.data[i % len(self.data)])
      res +=']'
      return res
```

c) Sehr gründliche Implementation einer möglichen Testklasse. Beim Testen ist darauf zu achten, dass möglichst alle typischen Fälle abgedeckt werden und auch die Fehlerfälle (Vorbedingungen) geprüft werden.

```
import unittest, random

class TestDeque(unittest.TestCase):
    # Konstruktor und das Verhalten bei leerer Deque werden getestet
    def testConstructor(self):
        self.assertRaises(TypeError, Deque, "Hallo")
        self.assertRaises(ValueError, Deque, 0)
        self.assertRaises(ValueError, Deque, -1)
        q = Deque(2)
        self.assertEqual(q.capacity(), 2)
        self.assertEqual(q.size(), 0)
        self.assertRaises(RuntimeError, q.popFirst)
        self.assertRaises(RuntimeError, q.popLast)

    # push() wird getestet
    def testPush(self):
        for y in range(0,100):
            q = Deque()

            for i in range(0, y):
                q.push(i)
                self.assertEqual(q.size(), i+1)
                self.assertTrue(q.size() <= q.capacity())

    # Deque wird n-mal gepushed und mit popLast() (n+1)-mal gepopped.
    # Das letzte popLast() muss eine Exception ausloesen (leere Deque).
    def testPopLast(self):
        for y in range(0,100):
            q = Deque()
            for i in range(0, y):
                q.push(i)

            for i in range(y, 0, -1):
                self.assertEqual(q.size(), i)
                self.assertEqual(q.popLast(), i-1)

            self.assertEqual(q.size(), 0)
            self.assertRaises(RuntimeError, q.popLast)

    # Deque wird n-mal gepushed und mit popFirst() (n+1)-mal gepopped.
    # Das letzte popFirst() muss eine Exception ausloesen (leere Deque).
    def testPopFirst(self):
        for y in range(0,100):
            q = Deque()
            for i in range(0, y):
                q.push(i)

            for i in range(y, 0, -1):
                self.assertEqual(q.size(), i)
                self.assertEqual(q.popFirst(), y-i)

            self.assertEqual(q.size(), 0)
            self.assertRaises(RuntimeError, q.popFirst)

    # push(), popFirst() und popLast() werden in zufaelliger Kombination
    # getestet. Dies wird 10-mal wiederholt.
    # (Wir fuellen die Deque am Anfang mit 115 Elementen, so dass ein
    #  Fehler wegen leerer Deque praktisch ausgeschlossen ist.)
    def testRand(self):
        for k in xrange(10):
            q = Deque()
```

```
            sizeCount = 115
            for i in range(sizeCount):
                q.push(i)
            self.assertEqual(q.size(), sizeCount)

            for i in range(400):
                dec = random.randint(0,3)
                if dec <= 1:
                    q.push(None)
                    sizeCount += 1
                elif dec == 2:
                    q.popFirst()
                    sizeCount -= 1
                elif dec == 3:
                    q.popLast()
                    sizeCount -= 1
                self.assertEqual(q.size(), sizeCount)

    # Teste, ob die Implementation des Rings fehlerhaft ist.
    def testShiftRight(self):
        q = Deque()

        # 10 Elemente einfuegen
        for i in range(0,10):
            q.push(i)
        self.assertEqual(q.capacity(), 16)
        self.assertEqual(q.size(), 10)
        self.assertEqual(q.head, 0)
        self.assertEqual(q.tail, 10)

        # die ersten 6 Elemente entfernen
        for i in range(0,6):
            q.popFirst()
        self.assertEqual(q.size(), 4)
        self.assertLess(q.head, q.tail)

        # weitere 10 Elemente einfuegen => tail
        # ueberschreitet die Arraygrenze und ist jetzt
        # kleiner als head
        for i in range(0,10):
            q.push(i)
        self.assertEqual(q.capacity(), 16)
        self.assertEqual(q.size(), 14)
        self.assertGreater(q.head, q.tail)

        # 6 Elemente am Ende entfernen => tail
        # ueberschreitet die Arraygrenze in der anderen
        # Richtung und ist wieder groesser als head
        for i in range(0,6):
            q.popLast()
        self.assertEqual(q.capacity(), 16)
        self.assertEqual(q.size(), 8)
        self.assertLess(q.head, q.tail)

        # die uebrigen Elemente am Ende entfernen =>
        # Deque muss jetzt leer sein
        for i in range(0,8):
            q.popLast()
        self.assertEqual(q.capacity(), 16)
        self.assertEqual(q.size(), 0)
        self.assertEqual(q.head, q.tail)

        # weiteres Entfernen muss Exception ausloesen
        self.assertRaises(RuntimeError, q.popFirst)
        self.assertRaises(RuntimeError, q.popLast)
```

```
            # 14 neue Elemente einfuegen => tail
            # ueberschreitet die Arraygrenze erneut und
            # ist kleiner als head
            for i in range(0,14):
                q.push(i)
            self.assertEqual(q.capacity(), 16)
            self.assertEqual(q.size(), 14)
            self.assertGreater(q.head, q.tail)

            # 11 Elemente am Anfang entfernen => head
            # ueberschreitet die Arraygrenze ebenfalls und
            # ist nun wieder kleiner als tail
            for i in range(0,11):
                q.popFirst()
            self.assertEqual(q.capacity(), 16)
            self.assertEqual(q.size(), 3)
            self.assertLess(q.head, q.tail)

            # die uebrigen Elemente am Anfang entfernen =>
            # Deque muss jetzt leer sein
            for i in range(0,3):
                q.popFirst()
            self.assertEqual(q.capacity(), 16)
            self.assertEqual(q.size(), 0)
            self.assertEqual(q.head, q.tail)

            # weiteres Entfernen muss Exception ausloesen
            self.assertRaises(RuntimeError, q.popFirst)
            self.assertRaises(RuntimeError, q.popLast)

            # 20 neue Elemente einfuegen => das interne Array wird
            # verdoppelt, head und tail daher zurueckgesetzt
            for i in range(0,20):
                q.push(i)
            self.assertEqual(q.capacity(), 32)
            self.assertEqual(q.size(), 20)
            self.assertEqual(q.head, 0)
            self.assertEqual(q.tail, 20)

    if __name__ == "__main__":
      print "Running Deque examples:"
      test =Deque(3)
      test.push(1)
      test.push(2)
      test.push(3)
      print "before:", test, "popFirst:", test.popFirst(), "after:", test
      test.push(4)
      print "before:", test, "popFirst:", test.popFirst(), "after:", test
      test.push(5)
      print "before:", test, "popFirst:", test.popFirst(), "after:", test
      test.push(6)
      print "before:", test, "popLast:", test.popLast(), "after:", test
      print "before:", test, "popLast:", test.popLast(), "after:", test
      print "before:", test, "popLast:", test.popLast(), "after:", test
      try:
        print "before:", test, "popLast:", test.popLast(), test
      except:
        print "Deque empty!"

      print "Running Deque unittests:"
      unittest.main()
```

Ausgabe des Programms:

```
Running Deque examples:
```

```
before: [1, 2, 3] popFirst: 1 after: [2, 3]
before: [2, 3, 4] popFirst: 2 after: [3, 4]
before: [3, 4, 5] popFirst: 3 after: [4, 5]
before: [4, 5, 6] popLast: 6 after: [4, 5]
before: [4, 5] popLast: 5 after: [4]
before: [4] popLast: 4 after: []
before: [] popLast: Deque empty!
Running Deque unittests:
......
----------------------------------------------------------------------
Ran 6 tests in 0.120s

OK
```