SWO3	Übung zu Softwareentwicklung mit klassischen Sprachen und Bibliotheken 3	WS 2018/19, Angabe 9

☒ Gruppe 1 (J. Heinzelreiter)

☐ Gruppe 2 (M. Hava) Name: Papesh Konstantin Aufwand [h]: 10

☐ Gruppe 3 (P. Kulczycki) Übungsleiter/Tutor: _____ Punkte: _____

Beispiel	Lösungsidee (max. 100%)	Implement. (max. 100%)	Testen (max. 100%)
1 (100 P)	90	90	80

Beispiel 1: swo3::deque (`src/deque/`)

Implementieren Sie einen ADT `swo3::deque` (*double-ended queue*, siehe https://en.wikipedia.org/wiki/Double-ended_queue) gemäß dem im Folgenden definierten Interface. Eine swo3::deque speichert ihre Elemente in einem Ringpuffer (siehe https://en.wikipedia.org/wiki/Circular_buffer). Testen Sie ausführlich unter Zuhilfenahme von generischen Algorithmen und *range-based for loops* (siehe https://en.cppreference.com/w/cpp/language/range-for). Für eine genaue Spezifikation der einzelnen Komponenten der `swo3::deque` (Typen, Methoden etc.) verweisen wir auf https://en.cppreference.com/w/cpp/container/deque und https://en.cppreference.com/w/cpp/named_req/RandomAccessIterator.

```
namespace swo3 {

/**
 * see https://en.cppreference.com/w/cpp/container/deque and
 *     https://en.cppreference.com/w/cpp/named_req/RandomAccessIterator
 */
template <typename T> class deque final {
   using value_type = ...
   using reference  = ...
   using size_type  = ...

   class iterator final {   // implements RandomAccessIterator
      ...
   };

   deque ();
   explicit deque (size_type count);
   deque (size_type count, T const & value);

   deque (deque const & other);
   deque (deque && other);
   deque (std::initializer_list <T> init);

  ~deque ();

   deque & operator = (deque const & other);
   deque & operator = (deque && other) noexcept;
   deque & operator = (std::initializer_list <T> init);

   reference operator [] (size_type pos);

   reference at    (size_type pos);
   reference back  ();
   reference front ();
```

```
   iterator begin () noexcept;
   iterator end   () noexcept;

   bool      empty () const noexcept;
   size_type size  () const noexcept;

   void clear () noexcept;

   void push_back (T const & value);
   void push_back (T && value);
   void pop_back  ();

   void push_front (T const & value);
   void push_front (T && value);
   void pop_front  ();

   void resize (size_type count);
   void swap   (deque & other) noexcept;

   ...
};

template <typename T> bool operator == (deque const & lhs, deque const & rhs);
template <typename T> bool operator != (deque const & lhs, deque const & rhs);
template <typename T> bool operator <  (deque const & lhs, deque const & rhs);
template <typename T> bool operator <= (deque const & lhs, deque const & rhs);
template <typename T> bool operator >  (deque const & lhs, deque const & rhs);
template <typename T> bool operator >= (deque const & lhs, deque const & rhs);

}   // namespace swo3
```

SWO31 & SWB31 Softwareentwicklung mit klassischen Sprachen und Bibliotheken – WS 2018/19
Übungsabgabe 9

Konstantin Papesh

13. Januar 2019

9.1 swo3::deque

9.1.1 Lösungsidee

Es ist eine Deque zu implementieren, welche auf den Spezifikationen der std::deque basiert. Weiters ist ein RandomAccessIterator dafür zu implementieren.

deque

Eine Deque ist grundsätzlich ein Ringpuffer, wobei die swo3::deque als Zusatzfunktion die automatische Erweiterung bei Speicherplatzmangel bietet. Sobald die Deque voll ist wird der Speicherplatz verdoppelt, um Platz für die weiteren Werte zu machen. Auch sollte die Deque generisch programmiert werden, um das Einfügen eines beliebigen Datentyps zu ermöglichen.
Ausserdem wird am Anfang der Klasse mithilfe von *using* ein Alias für alle Datentypen festgelegt. Dies ermöglicht eine vereinfachte Abänderung dieser. Danach werden die Konstruktoren definiert und zugleich implementiert. Da in swo3::deque mit Templates gearbeitet wird, muss die Implementierung direkt in der Header-Datei erfolgen. Wird der Konstruktor ohne Parameter aufgerufen, wird eine Deque mit der Standartgröße erstellt. Weiters wird die Konstruktion mit einer Size, einer Size + einzufügendem Wert, einer anderen Deque, einer anderen Deque mithilfe von std::move und einer Initializer-List ermöglicht.
Im Destruktor selbst wird nur der Pointer auf das Datenarray gelöscht, dies geschied mithilfe der Methode *.reset()*, da mit dem in der Standartbibliothek enthaltenen *unique_ptr* arbeitet wird.
Darauffolgend werden die Zuweisungsoperatoren überladen, auch hier wird die Möglichkeit einer Copy-, Move- oder List-Assignment geboten. Danach werden zwei Funktionen implementiert, um auf ein beliebiges Element des Puffers zugreifen zu können. Dabei unterscheiden sich *[]* und *at* dadurch, dass bei *at* Bounds-Checking durgeführt wird, daher es wird darauf geachtet dass der angeforderte Index auch innerhalb des Buffers liegt, ansonsten wird eine Exception geworfen. Als Laufzeitoptimierung wird bei *[]* kein

Bounds-Checking durchgeführt.
Danach werden Funktionen für das Abholen des ersten und letzten Elementes angeboten.
Die Methoden *begin* und *end* sind für *ranged-based for loops* wichtig, da diese für diese Funktionalität benötigt werden. Diese Funktionen liefern jeweils einen Iterator zurück, mit welchem über den Puffer iteriert werden kann.
Die Methode pop_[front/back] ermöglicht es, ein Element am Anfang/Ende des Buffers zu löschen.
Die Methode push_[front/back] ermöglicht es hingegen, ein Element hinzuzufügen. Dies kann entweder mit Copy oder Move passieren.
Letzendlich wird *resize* und *swap* implementiert. Resize wird automatisch aufgerufen sobald der Puffer voll ist. Dabei ist darauf zu achten, dass die zusätzlichen Speicherplätze zwischen dem Front- und End-Index frei werden, nur so können dann auch Werte hinzugefügt werden ohne die Reihenfolge des Buffers zu verändern oder die Datenintegrität zu verletzen.
Die Methode *swap* ermöglicht es, zwei gleich lange Deques zu vertauschen.

iterator

Der Iterator wird als interne Klasse in *deque* ausgeführt und ermöglicht die Verwendung dieser Iteratoren um auf Elemente zuzugreifen oder beispielsweise range-based for loops. Im Gegensatz zu Pointer wird der Zugriff auf das Element abstrahiert, daher kann gegen fehlerhaften Zugriffe vorgegangen werden. Beispielsweise ermöglichen Iteratoren uns das einfache Iterieren über die gesamte Pufferlänge, ohne dass sich der Nutzer sorgen um das Ende des zugewiesenen Speichers und den damit verbundenen *loop around* machen muss.
Der Iterator besitzt zwei Konstruktoren, einen ohne Parameter und einen Copy-Constructor. Dabei ergibt der Konstruktor ohne Parameter nur Sinn, wenn die internen Daten von einer darüberliegenden Funktion von *swo:3:deque* eingefügt werden.
Weiters verfügt der implementierte Iterator über alle Operatoren des *RandomAccessIterators*, es können daher verschiedenene mathematische Aktionen wie *++* oder *!=* durchgeführt werden. Auch Größenvergleiche und Dereferenzieren ist möglich. Vorallem bei Operationen welche den 'Index' verändern ist es wichtig zu überprüfen ob ein *Loop Around* vonnöten ist, also dass der Index hinter dem Ende des Speicherbereichs liegen würde. Dann muss, wenn das Ende dieses erreicht ist, an den Anfang des Speicherbereichs gesprungen werden und dort fortgefahren werden. Dasselbe gilt wenn ein negativer Index adressiert werden würde.

namespace swo3

Ausserhalb der Klasse *deque* werden die Vergleichsoperatoren für die Deque implementiert, damit man verschiedene Deques untereinander vergleichen kann. Die Implementierung ausserhalb der Klasse wird dadurch ermöglicht, dass nur öffentliche Funktionen der Klasse verwenden werden, um die Vergleiche durchzuführen.

9.1.2 Implementierung

Listing 9.1: main.cpp

```
1  //
2  // Created by khp on 05.01.19.
3  //
4  #include <iostream>
5  #include "deque.h"
6
7  void test_constructor() {
8      std::cout << "### TEST CONSTRUCTION ###" << std::endl;
9      swo3::deque<char> deque1;
10     swo3::deque<int> deque2(8);
11     swo3::deque<int> deque3(4,0);
12 }
13 void test_pushPop() {
14     std::cout << "### TEST PUSH_POP ###" << std::endl;
15     swo3::deque<int> deque2(8);
16
17     deque2.push_back(2);
18     deque2.push_back(3);
19     deque2.push_back(4);
20     deque2.push_front(2);
21
22     std::cout << deque2.size() << " <-- should be 4" << std::endl;
23 }
24 void test_assign() {
25     std::cout << "### TEST ASSIGN ###" << std::endl;
26     swo3::deque<int> deque1(4);
27     swo3::deque<int> deque2;
28
29     deque2 = deque1;
30 }
31 void test_at() {
32     std::cout << "### TEST AT METHOD ###" << std::endl;
33     swo3::deque<int> deque1(4);
34
35     deque1.push_back(2);
36     deque1.push_back(3);
37     deque1.push_back(4);
38     deque1.push_front(2);
39
40     std::cout << deque1.at(0) << " <-- should be 2" << std::endl;
41     try {
42         std::cout << deque1.at(99) << " <-- should throw an error" << std::endl;
43     } catch(std::out_of_range& err) {
44         std::cout << err.what() << std::endl;
45     }
46 }
47 void test_index() {
48     std::cout << "### TEST INDEX ###" << std::endl;
49     swo3::deque<int> deque1(4);
50
51     deque1.push_back(7);
52     deque1.push_back(3);
53     deque1.push_back(4);
```

```
54      deque1.push_front(2);
55
56      std::cout << deque1.at(1) << " <-- should be 3" << std::endl;
57  }
58  void test_pushPopResize() {
59      std::cout << "### TEST PUSH_POP WITH RESIZE ###" << std::endl;
60      swo3::deque<int> deque2(3);
61
62      deque2.push_back(2);
63      deque2.push_back(3);
64      deque2.push_back(4);
65      deque2.push_front(2);
66      deque2.push_front(2);
67      deque2.push_front(2);
68
69      std::cout << deque2.size() << " <-- should be 6" << std::endl;
70  }
71  void test_iter_auto() {
72      std::cout << "### TEST ITERATOR AUTO ###" << std::endl;
73      swo3::deque<int> deque2(8);
74
75      deque2.push_back(2);
76      deque2.push_back(3);
77      deque2.push_back(4);
78
79      deque2.push_front(6);
80      deque2.push_front(22);
81      deque2.push_front(3);
82
83      for (const auto &item : deque2) {
84          std::cout << item << std::endl;
85      }
86  }
87  void test_iter() {
88      std::cout << "### TEST ITERATOR ###" << std::endl;
89      swo3::deque<int> deque2(8);
90
91      deque2.push_back(2);
92      deque2.push_back(3);
93      deque2.push_back(4);
94
95      deque2.push_front(6);
96      deque2.push_front(22);
97      deque2.push_front(3);
98
99      for(auto item = deque2.begin(); item != deque2.end(); ++item){
100         std::cout << *item << std::endl;
101     }
102 }
103 void test_frontBack() {
104     std::cout << "### TEST FRONT-BACK ###" << std::endl;
105     swo3::deque<int> deque1(8);
106
107     deque1.push_back(2);
108     deque1.push_back(3);
109     deque1.push_back(4);
110
```

```
111     std::cout << deque1.front() << " <-- should be 2" << std::endl;
112     std::cout << deque1.back() << " <-- should be 4" << std::endl;
113 }
114 void test_swap() {
115     std::cout << "### TEST SWAP ###" << std::endl;
116     swo3::deque<int> deque1(8);
117 
118     deque1.push_back(2);
119     deque1.push_back(3);
120     deque1.push_back(4);
121 
122     swo3::deque<int> deque2(8);
123 
124     deque2.push_front(6);
125     deque2.push_front(22);
126     deque2.push_front(3);
127 
128     std::cout << "## BEFORE SWAP ##" << std::endl;
129     std::cout << "# DEQUE1 #" << std::endl;
130     for (const auto &item : deque1) {
131         std::cout << item << std::endl;
132     }
133     std::cout << "# DEQUE2 #" << std::endl;
134     for (const auto &item2 : deque2) {
135         std::cout << item2 << std::endl;
136     }
137 
138     deque1.swap(deque2);
139 
140     std::cout << "## AFTER SWAP ##" << std::endl;
141     std::cout << "# DEQUE1 #" << std::endl;
142     for (const auto &item : deque1) {
143         std::cout << item << std::endl;
144     }
145     std::cout << "# DEQUE2 #" << std::endl;
146     for (const auto &item2 : deque2) {
147         std::cout << item2 << std::endl;
148     }
149 }
150 int main() {
151     test_constructor();
152     test_assign();
153     test_at();
154     test_index();
155     test_frontBack();
156     test_pushPop();
157     test_pushPopResize();
158     test_iter();
159     test_iter_auto();
160     test_swap();
161 
162     return EXIT_SUCCESS;
163 }
```

Listing 9.2: deque.h

```
1  //
2  // Created by khp on 05.01.19.
3  //
4
5  #ifndef SWO3_DEQUE_H
6  #define SWO3_DEQUE_H
7
8  #include <initializer_list>
9  #include <memory>
10 #include <iostream>
11
12 #define DEFAULT_SIZE 8
13 #define SIZE_MULTIPLIER 2
14
15 namespace swo3 {
16     template<typename T>
17     class deque final {
18         using value_type = T;
19         using reference = T &;
20         using pointer = T *;
21         using array = T[];
22         using pointerArr = std::unique_ptr<array>;
23         using size_type = size_t;
24
25         class iterator final : public std::iterator<std::random_access_iterator_tag,
       T> {
26             friend deque;
27         public:
28             using difference_type = int;
29
30             iterator() = default;
31
32             iterator(iterator const &other) : _pos{other._pos}, _first{other._first
       }, _last{other._last} {
33             };
34
35             iterator &operator=(iterator const &other) {
36                 _pos = other._pos;
37                 _first = other._first;
38                 _last = other._last;
39                 return *this;
40             };
41
42             iterator &operator++() {
43                 if (_pos == _last)  // loop around
44                     _pos = _first;
45                 else
46                     _pos = (_pos + 1);
47                 return *this;
48             }
49
50             iterator operator++(int) {
51                 iterator tIt = *this;
52                 if (_pos == _last)  // loop around
53                     _pos = _first;
```

```
54                  else
55                      _pos = (_pos + 1);
56                  return tIt;
57              }
58
59              bool operator==(iterator const &other) {
60                  return _pos == other._pos;
61              }
62
63              bool operator!=(iterator const &other) {
64                  return _pos != other._pos;
65              }
66
67              reference operator*() {
68                  return *_pos;
69              }
70
71              iterator &operator--() {
72                  if (_pos == _first)  // loop around
73                      _pos = _last;
74                  else
75                      _pos = (_pos - 1);
76                  return this;
77              }
78
79              iterator operator--(int) {
80                  iterator tIt = *this;
81                  if (_pos == _first)  // loop around
82                      _pos = _last;
83                  else
84                      _pos = (_pos - 1);
85                  return &tIt;
86              }
87
88              iterator &operator+=(difference_type n) {
89                  _pos += n;
90                  return &this;
91              }
92
93              iterator operator+(difference_type n) {
94                  iterator tIt;
95                  tIt._pos = _pos + n;
96                  return tIt;
97              }
98
99              iterator operator+(iterator const &other) {
100                 iterator tIt;
101                 tIt._pos = _pos + other._pos;
102                 return tIt;
103             }
104
105             iterator &operator-=(difference_type n) {
106                 _pos -= n;
107                 return &this;
108             }
109
110             iterator operator-(difference_type n) {
```

```
111                 iterator tIt;
112                 tIt._pos = _pos - n;
113                 return tIt;
114             }
115 
116             difference_type operator-(iterator const &other) {
117                 return _pos - other._pos;
118             }
119 
120             reference operator[](difference_type n) {
121                 iterator tIt = *this;
122                 for (difference_type i = 0; i < n; ++i) {
123                     if (tIt._pos == _last)
124                         throw std::out_of_range("index out of range!");
125                     tIt++;
126                 }
127                 return *tIt._pos;
128             }
129 
130             bool operator<(iterator const &other) {
131                 return _pos < other._pos;
132             }
133 
134             bool operator>(iterator const &other) {
135                 return _pos > other._pos;
136             }
137 
138             bool operator<=(iterator const &other) {
139                 return _pos <= other._pos;
140             }
141 
142             bool operator>=(iterator const &other) {
143                 return _pos >= other._pos;
144             }
145 
146         private:
147             pointer _pos;
148             pointer _first;
149             pointer _last;
150         };
151 
152     public:
153         deque() {
154             deque(DEFAULT_SIZE);
155         }
156 
157         explicit deque(size_type count) : _size{count}, _data{std::make_unique<array
    >(count)} {
158 
159         }
160 
161         deque(size_type count, T const &value) : _size{count}, _data{std::
    make_unique<array>(count)} {
162             for (size_type i = 0; i < _size; ++i) {
163                 _data[i] = value;
164             }
165             _end = _size - 1;
```

```
166         };
167
168         deque(deque const &other) : _size{other._size}, _data{std::make_unique<array
        >(other._size)},
169                                     _front{other._front}, _end{other._end} {
170             std::copy(other._data, other._data + _size, _data);
171         };
172
173         deque(deque &&other) noexcept : _size{other._size}, _data{other._data},
174                                         _front{other._front}, _end{other._end} {
175             other._size = 0;
176             other._data.reset();
177         };
178
179         deque(std::initializer_list<T> init) {
180             _size = init.size();
181             _front = 0;
182             _end = _size - 1;
183             _data = std::make_unique<array>(_size);
184             for (size_type i = 0; i < _size; ++i) {
185                 _data[i] = *(init.begin() + i);
186             }
187         };
188
189         ~deque() {
190             _data.reset();
191         };
192
193         deque &operator=(deque const &other) {
194             _front = other._front;
195             _end = other._end;
196             _size = other._size;
197             _data = std::make_unique<array>(_size);
198             std::copy(other._data.get(), other._data.get() + _size, _data.get());
199             return *this;
200         };
201
202         deque &operator=(deque &&other) noexcept {
203             _front = other._front;
204             _end = other._end;
205             _size = other._size;
206             _data = std::move(other._data);
207             other._size = 0;
208             other._data.reset();
209             return *this;
210         };
211
212         deque &operator=(std::initializer_list<T> init) {
213             _size = init.size();
214             _front = _size - 1;
215             _end = 0;
216             _data = std::make_unique<array>(_size);
217             for (size_type i = 0; i < _size; ++i) {
218                 _data[i] = init[i];
219             }
220             return *this;
221         }
```

```
222
223         reference operator[](size_type pos) {
224             return &_data[pos];
225         }
226
227         reference at(size_type pos) {
228             if (pos > _size) {
229                 throw std::out_of_range("index out of range!");
230             }
231             return _data[pos];
232         }
233
234         reference back() {
235             if (_end == _front) {
236                 throw std::range_error("Empty deque!");
237             }
238             return _data[_end-1];
239         }
240
241         reference front() {
242             if (_front == _end) {
243                 throw std::range_error("Empty deque!");
244             }
245             return _data[_front];
246         }
247
248         iterator begin() noexcept {
249             iterator tIt;
250             tIt._pos = &_data[_front];
251             tIt._first = &_data[0];
252             tIt._last = &_data[_size - 1];
253             return tIt;
254         }
255
256         iterator end() noexcept {
257             iterator tIt;
258             tIt._pos = &_data[_end];
259             tIt._first = &_data[0];
260             tIt._last = &_data[_size - 1];
261             return tIt;
262         }
263
264         bool empty() const noexcept {
265             return _front == _end;
266         }
267
268         size_type size() const noexcept {
269             if (_end < _front)
270                 return _size - (_front - _end);
271             else
272                 return _end + 1 - _front;
273         }
274
275         size_type max_size() const noexcept {
276             return _size;
277         };
278
```

```
279         void clear() noexcept {
280             _size = 0;
281             _front = 0;
282             _end = 0;
283             _data.reset();
284         }
285 
286         void push_back(T const &value) {
287             if ((_end + 1) % _size == _front) {
288                 resize(SIZE_MULTIPLIER * _size);
289             }
290             _data[_end] = value;
291             _end = (_end + 1) % _size; // loop around
292         }
293 
294         void push_back(T &&value) {
295             if ((_end + 1) % _size == _front) {
296                 resize(SIZE_MULTIPLIER * _size);
297             }
298             _data[_end] = std::move(value);
299             _end = (_end + 1) % _size; // loop around
300         }
301 
302         void pop_back() {
303             if (_end == _front) {
304                 std::cerr << "Nothing to pop!" << std::endl;
305                 return;
306             }
307             if (_end == 0)// loop around
308                 _end = _size - 1;
309             else
310                 _end = (_end - 1) % _size;
311         }
312 
313         void push_front(T const &value) {
314             if (_front == 0)// loop around
315                 _front = _size - 1;
316             else
317                 _front = (_front - 1) % _size;
318             if (_end == _front) {
319                 resize(SIZE_MULTIPLIER * _size);
320             }
321             _data[_front] = value;
322         }
323 
324         void push_front(T &&value) {
325             if (_front == 0)// loop around
326                 _front = _size - 1;
327             else
328                 _front = (_front - 1) % _size;
329             if (_end == _front) {
330                 resize(SIZE_MULTIPLIER * _size);
331             }
332             _data[_front] = std::move(value);
333         }
334 
335         void pop_front() {
```

```
336             if (_front == _end) {
337                 std::cerr << "Nothing to pop!" << std::endl;
338                 return;
339             }
340             _front = (_front + 1) % _size; // loop around
341         }
342 
343         void resize(size_type count) {
344             pointerArr tData = std::make_unique<array>(count);
345             std::copy(&_data[0], &_data[_end], tData.get());
346             std::copy(&_data[_front], &_data[_size], tData.get() + count - (_size -
    _front));
347             _front = count - (_size - _front);
348             _size = count;
349             _data = std::move(tData);
350         }
351 
352         void swap(deque &other) noexcept {
353             if (other._size != this->_size) {
354                 std::cerr << "Can't swap two different length deques" << std::endl;
355                 return;
356             }
357             deque tData;
358             tData = std::move(other);
359             other = std::move(*this);
360             *this = std::move(tData);
361         }
362 
363     private:
364         size_type _size{0};
365         pointerArr _data{nullptr};
366         size_type _front{0};
367         size_type _end{0};
368     };
369 
370     template<typename T>
371     bool operator==(deque<T> const &lhs, deque<T> const &rhs) {
372         return lhs.size() == rhs.size();
373     }
374 
375     template<typename T>
376     bool operator!=(deque<T> const &lhs, deque<T> const &rhs) {
377         return lhs.size() != rhs.size();
378     }
379 
380     template<typename T>
381     bool operator<(deque<T> const &lhs, deque<T> const &rhs) {
382         return lhs.size() < rhs.size();
383     }
384 
385     template<typename T>
386     bool operator<=(deque<T> const &lhs, deque<T> const &rhs) {
387         return lhs.size() <= rhs.size();
388     }
389 
390     template<typename T>
391     bool operator>(deque<T> const &lhs, deque<T> const &rhs) {
```

```
392         return lhs.size() > rhs.size();
393     }
394 
395     template<typename T>
396     bool operator>=(deque<T> const &lhs, deque<T> const &rhs) {
397         return lhs.size() >= rhs.size();
398     }
399 
400 }
401 
402 
403 #endif //SWO3_DEQUE_H
```

9.1.3 Testen

```
### TEST CONSTRUCTION ###
### TEST ASSIGN ###
### TEST AT METHOD ###
2 <-- should be 2
index out of range!
### TEST INDEX ###
3 <-- should be 3
### TEST FRONT-BACK ###
2 <-- should be 2
4 <-- should be 4
### TEST PUSH_POP ###
4 <-- should be 4
### TEST PUSH_POP WITH RESIZE ###
4 <-- should be 6
### TEST ITERATOR ###
3
22
6
2
3
4
### TEST ITERATOR AUTO ###
3
22
6
2
3
4
### TEST SWAP ###
## BEFORE SWAP ##
# DEQUE1 #
2
3
4
# DEQUE2 #
3
22
6
## AFTER SWAP ##
# DEQUE1 #
3
22
6
# DEQUE2 #
2
3
4
```

Abbildung 9.1: Verschiedene Testfälle