ADF2x & PRO2x Übungen zu Fortgeschrittenen Algorithmen & Datenstrukturen und OOP **SS 24, Übung 4**

Abgabetermin: Sa, 11.05.2024

☒ **Gr. 1,** S. Schöberl, MSc **Name** Elias Leonhardsberger **Aufwand in h** 9

☐ **Gr. 2,** DI (FH) G. Horn-Völlenkle, MSc

Punkte _____ **Tutor*in / Übungsleiter*in** ____ / ____

1. „Behälter“ *Vector* als ADS und ADT (12 + 6 Punkte)

Aus dem ersten Semester wissen Sie, dass man schon mit (Standard-)Pascal Felder auch dynamisch anlegen kann, womit es möglich ist, die Größe eines Felds erst zur Laufzeit zu fixieren. Hier ein einfaches Beispiel:

```
TYPE
  IntArray = ARRAY [1..1] OF INTEGER;
VAR
  ap: ^IntArray; (* array pointer = pointer to dynamic array *)
  n, i: INTEGER;
BEGIN
  n := …; (* size of array *)
  GetMem(ap, n * SizeOf(INTEGER));
  IF ap = NIL THEN … (* report heap overflow error and … *)
  FOR i := 1 TO n DO BEGIN
    (*$R-*)
    ap^[i] := 0;
    (*$R+*)
  END; (* FOR *)
  …
  FreeMem(ap, n * SizeOf(INTEGER));
```

Das Problem dabei ist, dass ein Feld immer noch mit einer bestimmten Größe (wenn auch erst zur Laufzeit) angelegt wird und sich diese Größe später (bei der Verwendung) nicht mehr ändern lässt.

Man kann aber auf der Basis von dynamischen Feldern einen wesentlich flexibleren „Behälter“ (engl. *collection*) mit der üblichen Bezeichnung `Vector` bauen, der seine Größe zur Laufzeit automatisch an die Bedürfnisse der jeweiligen Anwendung anpasst, in dem ein `Vector` zu Beginn zwar nur Platz für eine bestimmte Anzahl von Elementen (z. B. für 10) bietet, wenn diese Größe aber nicht ausreichen sollte, seine Größe automatisch anpasst, indem er ein neues, größeres (z. B. doppelt so großes) Feld anlegt, sämtliche Einträge vom alten in das neue Feld kopiert und dann das alte Feld freigibt.

a) Implementieren Sie einen Behälter `Vector` für Elemente des Typs `INTEGER` als abstrakte Datenstruktur (in Form eines Moduls `VADS.pas`), die mindestens folgende Operationen bietet:

`PROCEDURE Add(val: INTEGER);`
fügt den Wert `val` „hinten“ an, wobei zuvor ev. die Größe des Behälters angepasst wird.

`PROCEDURE SetElementAt(pos: INTEGER; val: INTEGER);`
setzt an der Stelle `pos` den Wert `val`.

`FUNCTION ElementAt(pos: INTEGER): INTEGER;`
liefert den Wert an der Stelle `pos`.

`PROCEDURE RemoveElementAt(pos: INTEGER);`
entfernt den Wert an der Stelle `pos`, wobei die restlichen Elemente um eine Position nach „vorne“ verschoben werden, die Kapazität des Behälters aber unverändert bleibt.

`FUNCTION Size: INTEGER;`
liefert die aktuelle Anzahl der im Behälter gespeicherten Werte (zu Beginn 0).

`FUNCTION Capacity: INTEGER;`
liefert die Kapazität des Behälters, also die aktuelle Größe des dynamischen Felds.

Achten Sie bei der Implementierung obiger Operationen darauf, alle Fehlersituationen zu erkennen, diese zu melden und passend zu behandeln.

b) Bauen Sie Ihre Lösung aus a) in einen abstrakten Daten**typ** (in Form eines Moduls `VADT.pas`) um, sodass nun auch mehrere Exemplare eines `Vectors` erstellt werden können. Achten Sie darauf, dass die Schnittstelle des dabei implementierten Moduls möglichst keine Informationen über die Implementierung des abstrakten Datentyps preisgibt.

2. Jetzt eine Warteschlange (*Queue*) (6 Punkte)

Bauen Sie unter geschickter Verwendung der für 1.a) entwickelten abstrakten Daten**struktur** ein Modul für eine neue abstrakte Daten**struktur**, welche eine Warteschlange realisiert (in `QADS.pas`).

Es sind mindestens folgende Operationen zur Verfügung zu stellen: `IsEmpty`, `Enqueue` (Element hinten einfügen) und `Dequeue` (Element vorne entfernen).

Hinweise:

1. Geben Sie für alle Ihre Lösungen immer eine „Lösungsidee“ an.
2. Dokumentieren und kommentieren Sie Ihre Algorithmen.
3. Bei Programmen: Geben Sie immer auch Testfälle ab, an denen man erkennen kann, dass Ihr Programm funktioniert, und dass es auch in Fehlersituation entsprechend reagiert.

ADF2/PRO2 UE04

Elias Leonhardsberger

8. Mai 2024, Hagenberg

Inhaltsverzeichnis

1 Vektor als Abstrakte Datenstruktur

1.1 Lösungsidee

Die Implementierung von ADS und ADT sind sich sehr ähnlich, daher wird nur die Idee des ADS beschrieben und dann zu ADT angepasst.

Um mithilfe von Add einen Werten "hintenänzuhängen wird er an dem ersten freien Index gespeichert. Der letzte Index wird als Size gespeichert. Wenn die Kapazität überschritten wird, muss das Array erweitert werden.

Die Erweiterung wird schon in der Angabe vorgegeben, es wird ein größeres Array angelegt und jeder Wert wird dort hineinkopiert.

SetElementAt setzt das Element an einem Index auf den mitgegebenen Wert. Durch die Anforderung der RemoveElementAt Prozedur wird angenommen, dass man damit nur Werte bearbeiten kann und keine hinzugefügt werden.

ElementAt gibt den Wert an dem gegebenen Index zurück, wenn kein Element am Index existiert wird mithilfe eines Rückgabewertes ein *FALSE* zurückgegeben. Durch diesen Rückgabeparameter, wird ElementAt zu einer Prozedur.

RemoveElementAt verringert die Size um 1, rückt die Werte nach dem gegebenen Index um eines nach vorne und verdrängt dadurch den Wert. Die Behandlung nicht existenter Elemente ist gleich wie bei ElementAt.

Size und Capacity geben einfach gespeicherte Werte des Vektors zurück.

Die Testfälle der einzelnen Units werden an Unit Tests angelehnt. Um den Zustand wieder zurückzusetzen, wird eine Clear Prozedur zum Interface hinzugefügt. Diese ist kein Dispose da die ADS kein Dispose benötigt.

1.2 Souce Code

1.2.1 VADS.pas

```
1
2  UNIT VADS;
3
4  INTERFACE
5
6  (* Adds an element to the end of the vector. If the vector is full, it
   ↪  will be resized. *)
7  (*  val - The value to be added to the vector. *)
8  (*  ok - A boolean value that will be set to TRUE if the operation was
   ↪  successful, FALSE otherwise. *)
9  PROCEDURE Add(val: INTEGER; VAR ok: BOOLEAN);
10
11 (* Sets the value of the element at the specified position. If the
   ↪  position is greater than the size of the vector, the operation will
   ↪  fail. *)
12 (*  pos - The position of the element to be set. *)
13 (*  val - The value to be set. *)
14 (*  ok - A boolean value that will be set to TRUE if the operation was
   ↪  successful, FALSE otherwise. *)
```

```
15 PROCEDURE SetElementAt(pos, val: INTEGER; VAR ok: BOOLEAN);
16
17 (* Gets the value of the element at the specified position. If the
   ↪  position is greater than the size of the vector, the operation will
   ↪  fail. *)
18 (*  pos - The position of the element to be retrieved. *)
19 (*  val - The value of the element at the specified position. *)
20 (*  ok - A boolean value that will be set to TRUE if the operation was
   ↪  successful, FALSE otherwise. *)
21 PROCEDURE ElementAt(pos: INTEGER; VAR val: INTEGER; VAR ok: BOOLEAN);
22
23 (* Removes the element at the specified position. If the position is
   ↪  greater than the size of the vector, the operation will fail. *)
24 (*  pos - The position of the element to be removed. *)
25 (*  ok - A boolean value that will be set to TRUE if the operation was
   ↪  successful, FALSE otherwise. *)
26 PROCEDURE RemoveElementAt(pos: INTEGER; VAR ok: BOOLEAN);
27
28 (* Returns the number of elements in the vector. *)
29 FUNCTION Size: INTEGER;
30
31 (* Returns the current capacity of the vector. *)
32 FUNCTION Capacity: INTEGER;
33
34 (* Clears the vector and sets its capacity to the base value. *)
35 PROCEDURE Clear;
36
37 IMPLEMENTATION
38
39 CONST
40   BASE_CAPACITY = 10;
41
42 TYPE
43   DynamicArray = ARRAY[1..1] OF INTEGER;
44
45 VAR
46   vSize, vCapacity: INTEGER;
47   dynArray: ^DynamicArray;
48
49 PROCEDURE ResizeVector(VAR ok: BOOLEAN);
50 VAR
51   i, oldCapacity: INTEGER;
52   newArray: ^DynamicArray;
53 BEGIN (* ResizeVector *)
54   oldCapacity := vCapacity;
55   vCapacity := oldCapacity * 2;
56   GetMem(newArray, vCapacity * sizeof(INTEGER));
```

```
57
58    IF (newArray = NIL) THEN
59      BEGIN (* IF *)
60        ok := FALSE;
61      END (* IF *)
62    ELSE
63      BEGIN (* ELSE *)
64        ok := TRUE;
65
66        FOR i := 1 TO oldCapacity DO
67          BEGIN (* FOR *)
68            {$R-}
69            newArray^[i] := dynArray^[i];
70            {$R+}
71          END; (* FOR *)
72
73        FreeMem(dynArray, oldCapacity * sizeof(INTEGER));
74        dynArray := newArray;
75      END; (* ELSE *)
76  END; (* ResizeVector *)
77
78  PROCEDURE Add(val: INTEGER; VAR ok: BOOLEAN);
79  BEGIN (* Add *)
80    vSize := vSize + 1;
81    ok := TRUE;
82
83    IF (vSize > vCapacity) THEN
84      BEGIN (* IF *)
85        ResizeVector(ok);
86      END; (* IF *)
87
88    IF (ok) THEN
89      BEGIN (* IF *)
90        {$R-}
91        dynArray^[vSize] := val;
92        {$R+}
93      END; (* IF *)
94  END; (* Add *)
95
96  PROCEDURE SetElementAt(pos, val: INTEGER; VAR ok: BOOLEAN);
97  BEGIN (* SetElementAt *)
98    IF (pos > vSize) OR (pos < 1) THEN
99      BEGIN (* IF *)
100       ok := FALSE
101     END (* IF *)
102   ELSE
103     BEGIN (* ELSE *)
```

```
104         ok := TRUE;
105         {$R-}
106         dynArray^[pos] := val;
107         {$R+}
108       END; (* ELSE *)
109   END; (* SetElementAt *)
110
111   PROCEDURE ElementAt(pos: INTEGER; VAR val: INTEGER; VAR ok: BOOLEAN);
112   BEGIN (* ElementAt *)
113     IF (pos > vSize) OR (pos < 1) THEN
114       BEGIN (* IF *)
115         ok := FALSE
116       END (* IF *)
117     ELSE
118       BEGIN (* ELSE *)
119         ok := TRUE;
120         {$R-}
121         val := dynArray^[pos];
122         {$R+}
123       END; (* ELSE *)
124   END; (* ElementAt *)
125
126   PROCEDURE RemoveElementAt(pos: INTEGER; VAR ok: BOOLEAN);
127   VAR
128     i: INTEGER;
129   BEGIN (* RemoveElementAt *)
130     IF (pos > vSize) OR (pos < 1) THEN
131       BEGIN (* IF *)
132         ok := FALSE
133       END (* IF *)
134     ELSE
135       BEGIN (* ELSE *)
136         ok := TRUE;
137         vSize := vSize - 1;
138
139         FOR i := pos TO vSize DO
140           BEGIN (* FOR *)
141             {$R-}
142             dynArray^[i] := dynArray^[i+1];
143             {$R+}
144           END; (* FOR *)
145       END; (* ELSE *)
146   END; (* RemoveElementAt *)
147
148   FUNCTION Size: INTEGER;
149   BEGIN (* Size *)
150     Size := vSize;
```

```
151 END; (* Size *)
152
153 FUNCTION Capacity: INTEGER;
154 BEGIN (* Capacity *)
155   Capacity := vCapacity;
156 END; (* Capacity *)
157
158 PROCEDURE Clear;
159 BEGIN (* Clear *)
160   IF (vCapacity <> BASE_CAPACITY) THEN
161     BEGIN
162       FreeMem(dynArray, vCapacity * sizeof(INTEGER));
163       vCapacity := BASE_CAPACITY;
164       GetMem(dynArray, vCapacity * sizeof(INTEGER));
165     END;
166
167   vSize := 0;
168 END; (* Clear *)
169
170 BEGIN (* VADS *)
171   vSize := 0;
172   vCapacity := BASE_CAPACITY;
173   GetMem(dynArray, vCapacity * sizeof(INTEGER));
174 END. (* VADS *)
```

1.3 Tests

1.3.1 TestVADS.pas

```
1  PROGRAM TestVADS;
2
3  USES
4  VADS;
5
6  TYPE
7    test = PROCEDURE (VAR success: BOOLEAN);
8
9  PROCEDURE InitialVector_IsEmpty(VAR success: BOOLEAN);
10 BEGIN (* InitialVector_IsEmpty *)
11   success := (Size() = 0)
12              AND (Capacity() = 10);
13 END; (* InitialVector_IsEmpty *)
14
15 PROCEDURE Clear_EmptiesVector(VAR success: BOOLEAN);
16 VAR
17   addOk: BOOLEAN;
18 BEGIN (* Clear_EmptiesVector *)
19   Add(1, addOk);
20   Clear();
21
22   success := (Size() = 0)
23              AND (Capacity() = 10)
24              AND addOk;
25 END; (* Clear_EmptiesVector *)
26
27 PROCEDURE AddingElement_IncreasesSize(VAR success: BOOLEAN);
28 VAR
29   addOk, elementAtOk: BOOLEAN;
30   elementAtValue: INTEGER;
31 BEGIN (* AddingElement_IncreasesSize *)
32   Clear();
33   Add(17, addOk);
34   ElementAt(1, elementAtValue, elementAtOk);
35
36   success := (Size() = 1)
37              AND (Capacity() = 10)
38              AND addOk
39              AND elementAtOk
40              AND (elementAtValue = 17);
41 END; (* AddingElement_IncreasesSize *)
42
43 PROCEDURE AddingElementsOverCapacity_ResizesVector(VAR success: BOOLEAN);
44 VAR
```

```
45    addOk, elementAtOk, ok: BOOLEAN;
46    i, elementAtValue: INTEGER;
47  BEGIN (* AddingElementsOverCapacity_ResizesVector *)
48    Clear();
49    i := 1;
50    ok := TRUE;
51
52    WHILE (ok) AND (i <= 10) DO
53      BEGIN (* WHILE *)
54        Add(i, addOk);
55        ElementAt(i, elementAtValue, elementAtOk);
56
57        ok := (Size() = i)
58              AND (Capacity() = 10)
59              AND addOk
60              AND elementAtOk
61              AND (elementAtValue = i);
62
63        i := i + 1;
64      END; (* WHILE *)
65
66    Add(11, addOk);
67    ElementAt(11, elementAtValue, elementAtOk);
68
69    success := ok
70               AND (Size() = 11)
71               AND (Capacity() = 20)
72               AND addOk
73               AND elementAtOk
74               AND (elementAtValue = 11);
75  END; (* AddingElementsOverCapacity_ResizesVector *)
76
77  PROCEDURE SetElementAt_UpdatesValue(VAR success: BOOLEAN);
78  VAR
79    addOk, elementAtOk, setElementAtOk, ok: BOOLEAN;
80    i, elementAtValue: INTEGER;
81  BEGIN (* SetElementAt_UpdatesValue *)
82    Clear();
83    i := 1;
84    ok := TRUE;
85
86    WHILE (ok) AND (i <= 10) DO
87      BEGIN (* WHILE *)
88        Add(i, addOk);
89        ElementAt(i, elementAtValue, elementAtOk);
90
91        ok := (Size() = i)
```

```
92              AND (Capacity() = 10)
93              AND addOk
94              AND elementAtOk
95              AND (elementAtValue = i);
96
97        i := i + 1;
98      END; (* WHILE *)
99
100   SetElementAt(5, 17, setElementAtOk);
101   ElementAt(5, elementAtValue, elementAtOk);
102
103   success := ok
104              AND setElementAtOk
105              AND (Size() = 10)
106              AND (Capacity() = 10)
107              AND elementAtOk
108              AND (elementAtValue = 17);
109 END; (* SetElementAt_UpdatesValue *)
110
111 PROCEDURE SetElementAtOutOfRange_ReturnsFalse(VAR success: BOOLEAN);
112 VAR
113   setElementAtOk: BOOLEAN;
114 BEGIN (* SetElementAtOutOfRange_ReturnsFalse *)
115   Clear();
116   SetElementAt(5, 17, setElementAtOk);
117
118   success := NOT setElementAtOk
119              AND (Size() = 0)
120              AND (Capacity() = 10);
121 END; (* SetElementAtOutOfRange_ReturnsFalse *)
122
123 PROCEDURE ElementAtOutOfRange_ReturnsFalse(VAR success: BOOLEAN);
124 VAR
125   elementAtOk: BOOLEAN;
126   elementAtValue: INTEGER;
127 BEGIN (* ElementAtOutOfRange_ReturnsFalse *)
128   Clear();
129   ElementAt(5, elementAtValue, elementAtOk);
130
131   success := NOT elementAtOk
132              AND (Size() = 0)
133              AND (Capacity() = 10);
134 END; (* ElementAtOutOfRange_ReturnsFalse *)
135
136 PROCEDURE RemoveElementAtOutOfRange_ReturnsFalse(VAR success: BOOLEAN);
137 VAR
138   removeElementAtOk: BOOLEAN;
```

```
139 BEGIN (* RemoveElementAtOutOfRange_ReturnsFalse *)
140   Clear();
141   RemoveElementAt(5, removeElementAtOk);
142 
143   success := NOT removeElementAtOk
144              AND (Size() = 0)
145              AND (Capacity() = 10);
146 END; (* RemoveElementAtOutOfRange_ReturnsFalse *)
147 
148 PROCEDURE RemoveElementAt_RemovesValueAndDoesNotResize(VAR success:
    ↪  BOOLEAN);
149 VAR
150   addOk, elementAtOk, removeElementAtOk, ok: BOOLEAN;
151   i, j, elementAtValue: INTEGER;
152 BEGIN (* RemoveElementAt_RemovesValueAndDoesNotResize *)
153   Clear();
154   i := 1;
155   ok := TRUE;
156 
157   WHILE (ok) AND (i <= 10) DO
158     BEGIN (* WHILE *)
159       Add(i, addOk);
160       ElementAt(i, elementAtValue, elementAtOk);
161 
162       ok := (Size() = i)
163             AND (Capacity() = 10)
164             AND addOk
165             AND elementAtOk
166             AND (elementAtValue = i);
167 
168       i := i + 1;
169     END; (* WHILE *)
170 
171   Add(11, addOk);
172   RemoveElementAt(5, removeElementAtOk);
173 
174   i := 1;
175   j := 1;
176 
177   WHILE (ok) AND (i <= 10) DO
178     BEGIN (* WHILE *)
179       ElementAt(i, elementAtValue, elementAtOk);
180 
181       ok := elementAtOk
182             AND (elementAtValue = j);
183 
184       i := i + 1;
```

```
185         j := j + 1;
186
187         IF (i = 5) THEN
188           BEGIN (* IF *)
189             j := j + 1;
190           END; (* IF *)
191       END; (* WHILE *)
192
193     success := ok
194                AND removeElementAtOk
195                AND (Size() = 10)
196                AND (Capacity() = 20);
197   END;
198
199   PROCEDURE AddingTenThousandElements_ResizesVector(VAR success: BOOLEAN);
200   VAR
201     addOk, elementAtOk, ok: BOOLEAN;
202     i, expectedCapacity, elementAtValue: INTEGER;
203   BEGIN (* AddingTenThousandElements_ResizesVector *)
204     Clear();
205     i := 1;
206     ok := TRUE;
207     expectedCapacity := 10;
208
209     WHILE (ok) AND (i <= 10000) DO
210       BEGIN (* WHILE *)
211         Add(i, addOk);
212         ElementAt(i, elementAtValue, elementAtOk);
213
214         ok := (Size() = i)
215               AND (Capacity() = expectedCapacity)
216               AND addOk
217               AND elementAtOk
218               AND (elementAtValue = i);
219
220         i := i + 1;
221
222         IF (i > expectedCapacity) THEN
223           BEGIN (* IF *)
224             expectedCapacity := expectedCapacity * 2;
225           END; (* IF *)
226       END; (* WHILE *)
227
228     success := ok
229   END; (* AddingTenThousandElements_ResizesVector *)
230
231   PROCEDURE RunTest(NAME: STRING; t: test);
```

```
232 VAR
233   success: BOOLEAN;
234 BEGIN (* RunTest *)
235   t(success);
236   IF (success) THEN
237     BEGIN (* IF *)
238       WriteLn('PASSED - ', name)
239     END (* IF *)
240   ELSE
241     BEGIN (* ELSE *)
242       WriteLn('FAILED - ', name);
243       Halt(1);
244     END; (* ELSE *)
245 END; (* RunTest *)
246
247 BEGIN (* TestVADS *)
248   RunTest('InitialVector_IsEmpty', InitialVector_IsEmpty);
249   RunTest('Clear_EmptiesVector', Clear_EmptiesVector);
250   RunTest('AddingElement_IncreasesSize', AddingElement_IncreasesSize);
251   RunTest('AddingElementsOverCapacity_ResizesVector',
      ↪ AddingElementsOverCapacity_ResizesVector);
252   RunTest('SetElementAt_UpdatesValue', SetElementAt_UpdatesValue);
253   RunTest('SetElementAtOutOfRange_ReturnsFalse',
      ↪ SetElementAtOutOfRange_ReturnsFalse);
254   RunTest('ElementAtOutOfRange_ReturnsFalse',
      ↪ ElementAtOutOfRange_ReturnsFalse);
255   RunTest('RemoveElementAtOutOfRange_ReturnsFalse',
      ↪ RemoveElementAtOutOfRange_ReturnsFalse);
256   RunTest('RemoveElementAt_RemovesValueAndDoesNotResize',
      ↪ RemoveElementAt_RemovesValueAndDoesNotResize);
257   RunTest('AddingTenThousandElements_ResizesVector',
      ↪ AddingTenThousandElements_ResizesVector);
258   WriteLn('All tests passed');
259 END. (* TestVADS *)
```

1.4 Testergebnisse

```
elias@EliasLaptop:~/Repos/SEbaBB2/pascal/PascalWorkspace/UE4/bin$ ./TestVADS
PASSED - InitialVector_IsEmpty
PASSED - Clear_EmptiesVector
PASSED - AddingElement_IncreasesSize
PASSED - AddingElementsOverCapacity_ResizesVector
PASSED - SetElementAt_UpdatesValue
PASSED - SetElementAtOutOfRange_ReturnsFalse
PASSED - ElementAtOutOfRange_ReturnsFalse
PASSED - RemoveElementAtOutOfRange_ReturnsFalse
PASSED - RemoveElementAt_RemovesValueAndDoesNotResize
PASSED - AddingTenThousandElements_ResizesVector
All tests passed
```

Abbildung 1: Ausgabe des Testprogramms *TestVADS*

2 Vektor als Abstrakter Datentyp

2.1 Lösungsidee

Für den abstrakten Datentypen wird die abstrakte Datenstruktur kopiert und wie in der Übung angepasst.

Size und Capacity bleiben Funktionen obwohl die einen *VAR* Parameter besitzen, da keine Seiteneffekte bestehen und der Parameter nur *VAR* ist um das Element nicht zu kopiern.

Ein Test für die geichzeitige Verwendung von Vektoren wird zusätzlich hinzugefügt.

2.2 Souce Code

2.2.1 VADT.pas

```
1
2  UNIT VADT;
3
4  INTERFACE
5
6  TYPE
7    Vector = POINTER;
8
9  (* Initializes the vector. *)
10 (*  v - The vector to be initialized. *)
11 (*  ok - A boolean value that will be set to TRUE if the operation was
   ↪  successful, FALSE otherwise. *)
12 PROCEDURE InitVector(VAR v: Vector; VAR ok: BOOLEAN);
13
14 (* Disposes the vector. *)
15 (*  v - The vector to be disposed. *)
16 PROCEDURE DisposeVector(VAR v: Vector);
17
18 (* Adds an element to the end of the vector. If the vector is full, it
   ↪  will be resized. *)
19 (*  v - The vector to which the element will be added. *)
20 (*  val - The value to be added to the vector. *)
21 (*  ok - A boolean value that will be set to TRUE if the operation was
   ↪  successful, FALSE otherwise. *)
22 PROCEDURE Add(VAR v: Vector; val: INTEGER; VAR ok: BOOLEAN);
23
24 (* Sets the value of the element at the specified position. If the
   ↪  position is greater than the size of the vector, the operation will
   ↪  fail. *)
25 (*  v - The vector in which the element will be set. *)
26 (*  pos - The position of the element to be set. *)
27 (*  val - The value to be set. *)
```

```
28  (*  ok - A boolean value that will be set to TRUE if the operation was
    ↪  successful, FALSE otherwise. *)
29  PROCEDURE SetElementAt(VAR v: Vector; pos, val: INTEGER; VAR ok:
    ↪  BOOLEAN);
30
31  (* Gets the value of the element at the specified position. If the
    ↪  position is greater than the size of the vector, the operation will
    ↪  fail. *)
32  (*  v - The vector from which the element will be retrieved. *)
33  (*  pos - The position of the element to be retrieved. *)
34  (*  val - The value of the element at the specified position. *)
35  (*  ok - A boolean value that will be set to TRUE if the operation was
    ↪  successful, FALSE otherwise. *)
36  PROCEDURE ElementAt(VAR v: Vector; pos: INTEGER; VAR val: INTEGER; VAR
    ↪  ok: BOOLEAN);
37
38  (* Removes the element at the specified position. If the position is
    ↪  greater than the size of the vector, the operation will fail. *)
39  (*  v - The vector from which the element will be removed. *)
40  (*  pos - The position of the element to be removed. *)
41  (*  ok - A boolean value that will be set to TRUE if the operation was
    ↪  successful, FALSE otherwise. *)
42  PROCEDURE RemoveElementAt(VAR v: Vector; pos: INTEGER; VAR ok: BOOLEAN);
43
44  (* Returns the number of elements in the vector. *)
45  (*  v - The vector whose size will be returned. *)
46  FUNCTION Size(VAR v: Vector): INTEGER;
47
48  (* Returns the current capacity of the vector. *)
49  (*  v - The vector whose capacity will be returned. *)
50  FUNCTION Capacity(VAR v: Vector): INTEGER;
51
52  (* Clears the vector and sets its capacity to the base value. *)
53  (*  v - The vector to be cleared. *)
54  PROCEDURE Clear(VAR v: Vector);
55
56  IMPLEMENTATION
57
58  CONST
59    BASE_CAPACITY = 10;
60
61  TYPE
62    DynamicArray = ARRAY[1..1] OF INTEGER;
63    State = RECORD
64      vSize, vCapacity: INTEGER;
65      dynArray: ^DynamicArray;
66    END;
```

```
67    StatePtr = ^State;
68
69  PROCEDURE InitVector(VAR v: Vector; VAR ok: BOOLEAN);
70  VAR
71    state: StatePtr;
72  BEGIN (* InitVector *)
73    state := NEW(StatePtr);
74
75    IF (state = NIL) THEN
76      BEGIN (* IF *)
77        ok := FALSE;
78      END (* IF *)
79    ELSE
80      BEGIN (* ELSE *)
81        state^.vSize := 0;
82        state^.vCapacity := BASE_CAPACITY;
83        GetMem(state^.dynArray, state^.vCapacity * sizeof(INTEGER));
84        v := state;
85        ok := TRUE;
86      END (* ELSE *)
87  END; (* InitVector *)
88
89  PROCEDURE DisposeVector(VAR v: Vector);
90  VAR
91    state: StatePtr;
92  BEGIN (* DisposeVector *)
93    state := StatePtr(v);
94    FreeMem(state^.dynArray, state^.vCapacity * sizeof(INTEGER));
95    DISPOSE(state);
96  END; (* DisposeVector *)
97
98  PROCEDURE ResizeVector(state: StatePtr; VAR ok: BOOLEAN);
99  VAR
100   i, oldCapacity: INTEGER;
101   newArray: ^DynamicArray;
102 BEGIN (* ResizeVector *)
103   oldCapacity := state^.vCapacity;
104   state^.vCapacity := oldCapacity * 2;
105   GetMem(newArray, state^.vCapacity * sizeof(INTEGER));
106
107   IF (newArray = NIL) THEN
108     BEGIN (* IF *)
109       ok := FALSE;
110     END (* IF *)
111   ELSE
112     BEGIN (* ELSE *)
113       ok := TRUE;
```

```
114
115         FOR i := 1 TO oldCapacity DO
116           BEGIN (* FOR *)
117             {$R-}
118             newArray^[i] := state^.dynArray^[i];
119             {$R+}
120           END; (* FOR *)
121
122         FreeMem(state^.dynArray, oldCapacity * sizeof(INTEGER));
123         state^.dynArray := newArray;
124       END; (* ELSE *)
125   END; (* ResizeVector *)
126
127   PROCEDURE Add(VAR v: Vector; val: INTEGER; VAR ok: BOOLEAN);
128   VAR
129     state: StatePtr;
130   BEGIN (* Add *)
131     state := StatePtr(v);
132     state^.vSize := state^.vSize + 1;
133     ok := TRUE;
134
135     IF (state^.vSize > state^.vCapacity) THEN
136       BEGIN (* IF *)
137         ResizeVector(state, ok);
138       END; (* IF *)
139
140     IF (ok) THEN
141       BEGIN (* IF *)
142         {$R-}
143         state^.dynArray^[state^.vSize] := val;
144         {$R+}
145       END; (* IF *)
146   END; (* Add *)
147
148   PROCEDURE SetElementAt(VAR v: Vector; pos, val: INTEGER; VAR ok:
      ↪ BOOLEAN);
149   VAR
150     state: StatePtr;
151   BEGIN (* SetElementAt *)
152     state := StatePtr(v);
153
154     IF (pos > state^.vSize) OR (pos < 1) THEN
155       BEGIN (* IF *)
156         ok := FALSE
157       END (* IF *)
158     ELSE
159       BEGIN (* ELSE *)
```

```
160         ok := TRUE;
161         {$R-}
162         state^.dynArray^[pos] := val;
163         {$R+}
164       END; (* ELSE *)
165   END; (* SetElementAt *)
166
167   PROCEDURE ElementAt(VAR v: Vector; pos: INTEGER; VAR val: INTEGER; VAR
      ↪  ok: BOOLEAN);
168   VAR
169     state: StatePtr;
170   BEGIN (* ElementAt *)
171     state := StatePtr(v);
172
173     IF (pos > state^.vSize) OR (pos < 1) THEN
174       BEGIN (* IF *)
175         ok := FALSE
176       END (* IF *)
177     ELSE
178       BEGIN (* ELSE *)
179         ok := TRUE;
180         {$R-}
181         val := state^.dynArray^[pos];
182         {$R+}
183       END; (* ELSE *)
184   END; (* ElementAt *)
185
186   PROCEDURE RemoveElementAt(VAR v: Vector; pos: INTEGER; VAR ok: BOOLEAN);
187   VAR
188     i: INTEGER;
189     state: StatePtr;
190   BEGIN (* RemoveElementAt *)
191     state := StatePtr(v);
192
193     IF (pos > state^.vSize) OR (pos < 1) THEN
194       BEGIN (* IF *)
195         ok := FALSE
196       END (* IF *)
197     ELSE
198       BEGIN (* ELSE *)
199         ok := TRUE;
200         state^.vSize := state^.vSize - 1;
201
202         FOR i := pos TO state^.vSize DO
203           BEGIN (* FOR *)
204             {$R-}
205             state^.dynArray^[i] := state^.dynArray^[i+1];
```

```
206             {$R+}
207           END; (* FOR *)
208       END; (* ELSE *)
209   END; (* RemoveElementAt *)
210
211   FUNCTION Size(VAR v: Vector): INTEGER;
212   VAR
213     state: StatePtr;
214   BEGIN (* Size *)
215     state := StatePtr(v);
216     Size := state^.vSize;
217   END; (* Size *)
218
219   FUNCTION Capacity(VAR v: Vector): INTEGER;
220   VAR
221     state: StatePtr;
222   BEGIN (* Capacity *)
223     state := StatePtr(v);
224     Capacity := state^.vCapacity;
225   END; (* Capacity *)
226
227   PROCEDURE Clear(VAR v: Vector);
228   VAR
229     state: StatePtr;
230   BEGIN (* Clear *)
231     state := StatePtr(v);
232     state^.vSize := 0;
233
234     IF (state^.vCapacity <> BASE_CAPACITY) THEN
235       BEGIN
236         FreeMem(state^.dynArray, state^.vCapacity * sizeof(INTEGER));
237         state^.vCapacity := BASE_CAPACITY;
238         GetMem(state^.dynArray, state^.vCapacity * sizeof(INTEGER));
239       END;
240   END; (* Clear *)
241
242   END.
```

2.3 Tests

2.3.1 TestVADT.pas

```
1  PROGRAM TestVADT;
2
3  USES
4  VADT;
5
6  TYPE
7    test = PROCEDURE (VAR v: Vector; VAR success: BOOLEAN);
8
9  PROCEDURE InitialVector_IsEmpty(VAR v: Vector; VAR success: BOOLEAN);
10 BEGIN (* InitialVector_IsEmpty *)
11   success := (Size(v) = 0)
12              AND (Capacity(v) = 10);
13 END; (* InitialVector_IsEmpty *)
14
15 PROCEDURE Clear_EmptiesVector(VAR v: Vector; VAR success: BOOLEAN);
16 VAR
17   addOk: BOOLEAN;
18 BEGIN (* Clear_EmptiesVector *)
19   Add(v, 1, addOk);
20   Clear(v);
21
22   success := (Size(v) = 0)
23              AND (Capacity(v) = 10)
24              AND addOk;
25 END; (* Clear_EmptiesVector *)
26
27 PROCEDURE AddingElement_IncreasesSize(VAR v: Vector; VAR success:
   ↪  BOOLEAN);
28 VAR
29   addOk, elementAtOk: BOOLEAN;
30   elementAtValue: INTEGER;
31 BEGIN (* AddingElement_IncreasesSize *)
32   Add(v, 17, addOk);
33   ElementAt(v, 1, elementAtValue, elementAtOk);
34
35   success := (Size(v) = 1)
36              AND (Capacity(v) = 10)
37              AND addOk
38              AND elementAtOk
39              AND (elementAtValue = 17);
40 END; (* AddingElement_IncreasesSize *)
41
42 PROCEDURE AddingElementsOverCapacity_ResizesVector(VAR v: Vector; VAR
   ↪  success: BOOLEAN);
```

```
43  VAR
44    addOk, elementAtOk, ok: BOOLEAN;
45    i, elementAtValue: INTEGER;
46  BEGIN (* AddingElementsOverCapacity_ResizesVector *)
47    i := 1;
48    ok := TRUE;
49
50    WHILE (ok) AND (i <= 10) DO
51      BEGIN (* WHILE *)
52        Add(v, i, addOk);
53        ElementAt(v, i, elementAtValue, elementAtOk);
54
55        ok := (Size(v) = i)
56              AND (Capacity(v) = 10)
57              AND addOk
58              AND elementAtOk
59              AND (elementAtValue = i);
60
61        i := i + 1;
62      END; (* WHILE *)
63
64    Add(v, 11, addOk);
65    ElementAt(v, 11, elementAtValue, elementAtOk);
66
67    success := ok
68               AND (Size(v) = 11)
69               AND (Capacity(v) = 20)
70               AND addOk
71               AND elementAtOk
72               AND (elementAtValue = 11);
73  END; (* AddingElementsOverCapacity_ResizesVector *)
74
75  PROCEDURE SetElementAt_UpdatesValue(VAR v: Vector; VAR success: BOOLEAN);
76  VAR
77    addOk, elementAtOk, setElementAtOk, ok: BOOLEAN;
78    i, elementAtValue: INTEGER;
79  BEGIN (* SetElementAt_UpdatesValue *)
80    i := 1;
81    ok := TRUE;
82
83    WHILE (ok) AND (i <= 10) DO
84      BEGIN (* WHILE *)
85        Add(v, i, addOk);
86        ElementAt(v, i, elementAtValue, elementAtOk);
87
88        ok := (Size(v) = i)
89              AND (Capacity(v) = 10)
```

```
90              AND addOk
91              AND elementAtOk
92              AND (elementAtValue = i);
93
94        i := i + 1;
95      END; (* WHILE *)
96
97    SetElementAt(v, 5, 17, setElementAtOk);
98    ElementAt(v, 5, elementAtValue, elementAtOk);
99
100   success := ok
101              AND setElementAtOk
102              AND (Size(v) = 10)
103              AND (Capacity(v) = 10)
104              AND elementAtOk
105              AND (elementAtValue = 17);
106 END; (* SetElementAt_UpdatesValue *)
107
108 PROCEDURE SetElementAtOutOfRange_ReturnsFalse(VAR v: Vector; VAR success:
    ↪  BOOLEAN);
109 VAR
110   setElementAtOk: BOOLEAN;
111 BEGIN (* SetElementAtOutOfRange_ReturnsFalse *)
112   SetElementAt(v, 5, 17, setElementAtOk);
113
114   success := NOT setElementAtOk
115              AND (Size(v) = 0)
116              AND (Capacity(v) = 10);
117 END; (* SetElementAtOutOfRange_ReturnsFalse *)
118
119 PROCEDURE ElementAtOutOfRange_ReturnsFalse(VAR v: Vector; VAR success:
    ↪  BOOLEAN);
120 VAR
121   elementAtOk: BOOLEAN;
122   elementAtValue: INTEGER;
123 BEGIN (* ElementAtOutOfRange_ReturnsFalse *)
124   ElementAt(v, 5, elementAtValue, elementAtOk);
125
126   success := NOT elementAtOk
127              AND (Size(v) = 0)
128              AND (Capacity(v) = 10);
129 END; (* ElementAtOutOfRange_ReturnsFalse *)
130
131 PROCEDURE RemoveElementAtOutOfRange_ReturnsFalse(VAR v: Vector; VAR
    ↪  success: BOOLEAN);
132 VAR
133   removeElementAtOk: BOOLEAN;
```

```
134 BEGIN (* RemoveElementAtOutOfRange_ReturnsFalse *)
135   RemoveElementAt(v, 5, removeElementAtOk);
136
137   success := NOT removeElementAtOk
138              AND (Size(v) = 0)
139              AND (Capacity(v) = 10);
140 END; (* RemoveElementAtOutOfRange_ReturnsFalse *)
141
142 PROCEDURE RemoveElementAt_RemovesValueAndDoesNotResize(VAR v: Vector; VAR
    ↪  success: BOOLEAN);
143 VAR
144   addOk, elementAtOk, removeElementAtOk, ok: BOOLEAN;
145   i, j, elementAtValue: INTEGER;
146 BEGIN (* RemoveElementAt_RemovesValueAndDoesNotResize *)
147   i := 1;
148   ok := TRUE;
149
150   WHILE (ok) AND (i <= 10) DO
151     BEGIN (* WHILE *)
152       Add(v, i, addOk);
153       ElementAt(v, i, elementAtValue, elementAtOk);
154
155       ok := (Size(v) = i)
156             AND (Capacity(v) = 10)
157             AND addOk
158             AND elementAtOk
159             AND (elementAtValue = i);
160
161       i := i + 1;
162     END; (* WHILE *)
163
164   Add(v, 11, addOk);
165   RemoveElementAt(v, 5, removeElementAtOk);
166
167   i := 1;
168   j := 1;
169
170   WHILE (ok) AND (i <= 10) DO
171     BEGIN (* WHILE *)
172       ElementAt(v, i, elementAtValue, elementAtOk);
173
174       ok := elementAtOk
175             AND (elementAtValue = j);
176
177       i := i + 1;
178       j := j + 1;
179
```

```
180         IF (i = 5) THEN
181           BEGIN (* IF *)
182             j := j + 1;
183           END; (* IF *)
184       END; (* WHILE *)
185
186     success := ok
187                AND removeElementAtOk
188                AND (Size(v) = 10)
189                AND (Capacity(v) = 20);
190   END;
191
192   PROCEDURE AddingTenThousandElements_ResizesVector(VAR v: Vector; VAR
      ↪  success: BOOLEAN);
193   VAR
194     addOk, elementAtOk, ok: BOOLEAN;
195     i, expectedCapacity, elementAtValue: INTEGER;
196   BEGIN (* AddingTenThousandElements_ResizesVector *)
197     i := 1;
198     ok := TRUE;
199     expectedCapacity := 10;
200
201     WHILE (ok) AND (i <= 10000) DO
202       BEGIN (* WHILE *)
203         Add(v, i, addOk);
204         ElementAt(v, i, elementAtValue, elementAtOk);
205
206         ok := (Size(v) = i)
207               AND (Capacity(v) = expectedCapacity)
208               AND addOk
209               AND elementAtOk
210               AND (elementAtValue = i);
211
212         i := i + 1;
213
214         IF (i > expectedCapacity) THEN
215           BEGIN (* IF *)
216             expectedCapacity := expectedCapacity * 2;
217           END; (* IF *)
218       END; (* WHILE *)
219
220     success := ok
221   END; (* AddingTenThousandElements_ResizesVector *)
222
223   PROCEDURE RunTest(NAME: STRING; t: test);
224   VAR
225     success: BOOLEAN;
```

```
226   v: Vector;
227 BEGIN (* RunTest *)
228   InitVector(v, success);
229
230   IF (NOT success) THEN
231     BEGIN
232       WriteLn('FAILED to initialize vector');
233       Halt(1);
234     END
235   ELSE
236     BEGIN
237       t(v ,success);
238       DisposeVector(v);
239
240       IF (success) THEN
241         BEGIN (* IF *)
242           WriteLn('PASSED - ', name);
243         END (* IF *)
244       ELSE
245         BEGIN (* ELSE *)
246           WriteLn('FAILED - ', name);
247           Halt(1);
248         END; (* ELSE *)
249     END;
250 END; (* RunTest *)
251
252 PROCEDURE TestConcurrentExecution;
253 VAR
254   v1, v2: Vector;
255   success1, success2: BOOLEAN;
256 BEGIN (* TestConcurrentExecution *)
257   InitVector(v1, success1);
258   InitVector(v2, success2);
259
260   IF (success1 AND success2) THEN
261     BEGIN (* IF *)
262       RemoveElementAt_RemovesValueAndDoesNotResize(v1, success1);
263       SetElementAt_UpdatesValue(v2, success2);
264       DisposeVector(v1);
265       DisposeVector(v2);
266
267       IF (success1 AND success2) THEN
268         BEGIN (* IF *)
269           WriteLn('PASSED - Concurrent Execution');
270         END (* IF *)
271       ELSE
272         BEGIN (* ELSE *)
```

```
273           WriteLn('FAILED - Concurrent Execution');
274           Halt(1);
275         END; (* ELSE *)
276     END (* IF *)
277   ELSE
278     BEGIN (* ELSE *)
279       WriteLn('FAILED to initialize vectors');
280       Halt(1);
281     END; (* ELSE *)
282 END; (* TestConcurrentExecution *)
283
284 BEGIN (* TestVADT *)
285   RunTest('InitialVector_IsEmpty', InitialVector_IsEmpty);
286   RunTest('Clear_EmptiesVector', Clear_EmptiesVector);
287   RunTest('AddingElement_IncreasesSize', AddingElement_IncreasesSize);
288   RunTest('AddingElementsOverCapacity_ResizesVector',
    ↪  AddingElementsOverCapacity_ResizesVector);
289   RunTest('SetElementAt_UpdatesValue', SetElementAt_UpdatesValue);
290   RunTest('SetElementAtOutOfRange_ReturnsFalse',
    ↪  SetElementAtOutOfRange_ReturnsFalse);
291   RunTest('ElementAtOutOfRange_ReturnsFalse',
    ↪  ElementAtOutOfRange_ReturnsFalse);
292   RunTest('RemoveElementAtOutOfRange_ReturnsFalse',
    ↪  RemoveElementAtOutOfRange_ReturnsFalse);
293   RunTest('RemoveElementAt_RemovesValueAndDoesNotResize',
    ↪  RemoveElementAt_RemovesValueAndDoesNotResize);
294   RunTest('AddingTenThousandElements_ResizesVector',
    ↪  AddingTenThousandElements_ResizesVector);
295   TestConcurrentExecution();
296   WriteLn('All tests passed');
297 END. (* TestVADT *)
```

2.4 Testergebnisse

```
elias@EliasLaptop:~/Repos/SEbaBB2/pascal/PascalWorkspace/UE4/bin$ ./TestVADT
PASSED - InitialVector_IsEmpty
PASSED - Clear_EmptiesVector
PASSED - AddingElement_IncreasesSize
PASSED - AddingElementsOverCapacity_ResizesVector
PASSED - SetElementAt_UpdatesValue
PASSED - SetElementAtOutOfRange_ReturnsFalse
PASSED - ElementAtOutOfRange_ReturnsFalse
PASSED - RemoveElementAtOutOfRange_ReturnsFalse
PASSED - RemoveElementAt_RemovesValueAndDoesNotResize
PASSED - AddingTenThousandElements_ResizesVector
PASSED - Concurrent Execution
All tests passed
```

Abbildung 2: Ausgabe des Testprogramms *TestVADT*

3 Queue

3.1 Lösungsidee

Die Queue nutzt den Vektor als Basis.

Enqueue ruft ein Add auf, IsEmpty schaut ob die Size 0 ist und Dequeue ruft ElementAt(1) und RemoveElementAt(1) auf.

3.2 Souce Code

3.2.1 QADS.pas

```
1
2  UNIT QADS;
3
4  INTERFACE
5
6  FUNCTION IsEmpty: BOOLEAN;
7
8  PROCEDURE Enqueue(val: INTEGER; VAR ok: BOOLEAN);
9
10 PROCEDURE Dequeue(VAR val: INTEGER; VAR ok: BOOLEAN);
11
12 PROCEDURE ClearQueue;
13
14 IMPLEMENTATION
15
16 USES
17 VADS;
18
19 FUNCTION IsEmpty: BOOLEAN;
20 BEGIN (* IsEmpty *)
21   IsEmpty := Size() = 0;
22 END; (* IsEmpty *)
23
24 PROCEDURE Enqueue(val: INTEGER; VAR ok: BOOLEAN);
25 BEGIN (* Enqueue *)
26   Add(val, ok);
27 END; (* Enqueue *)
28
29 PROCEDURE Dequeue(VAR val: INTEGER; VAR ok: BOOLEAN);
30 BEGIN (* Dequeue *)
31   IF IsEmpty() THEN
32     BEGIN (* IF *)
33       ok := FALSE
34     END (* IF *)
35   ELSE
36     BEGIN (* ELSE *)
```

```
37        ElementAt(1, val, ok);
38        RemoveElementAt(1, ok);
39      END; (* ELSE *)
40  END; (* Dequeue *)
41
42  PROCEDURE ClearQueue;
43  BEGIN (* ClearQueue *)
44    Clear();
45  END; (* ClearQueue *)
46
47  END.
```

3.3 Tests

3.3.1 TestQADS.pas

```
1  PROGRAM TestQADS;
2
3  USES
4  QADS;
5
6  TYPE
7    test = PROCEDURE (VAR success: BOOLEAN);
8
9  PROCEDURE InitialQueue_IsEmpty(VAR success: BOOLEAN);
10 BEGIN (* InitialQueue_IsEmpty *)
11   success := IsEmpty();
12 END; (* InitialQueue_IsEmpty *)
13
14 PROCEDURE Clear_EmptiesQueue(VAR success: BOOLEAN);
15 VAR
16   addOk: BOOLEAN;
17 BEGIN (* Clear_EmptiesVector *)
18   Enqueue(1, addOk);
19   ClearQueue();
20
21   success := IsEmpty()
22              AND addOk;
23 END; (* Clear_EmptiesVector *)
24
25 PROCEDURE Enqueue_AddsElement(VAR success: BOOLEAN);
26 VAR
27   enqueueOk, dequeueOk: BOOLEAN;
28   dequeueVal: INTEGER;
29 BEGIN (* Enqueue_AddsElement *)
30   ClearQueue();
31   Enqueue(1, enqueueOk);
32
33   success := enqueueOk
34              AND NOT IsEmpty();
35
36   Dequeue(dequeueVal, dequeueOk);
37
38   success := success
39              AND dequeueOk
40              AND (dequeueVal = 1)
41              AND IsEmpty();
42 END; (* Enqueue_AddsElement *)
43
44 PROCEDURE DequeueEmptyQueue_ReturnsFalse(VAR success: BOOLEAN);
```

```
45  VAR
46    dequeueOk: BOOLEAN;
47    dequeueVal: INTEGER;
48  BEGIN (* DequeueEmptyQueue_ReturnsFalse *)
49    ClearQueue();
50    Dequeue(dequeueVal, dequeueOk);
51
52    success := NOT dequeueOk
53               AND IsEmpty();
54  END; (* DequeueEmptyQueue_ReturnsFalse *)
55
56  PROCEDURE RunTest(NAME: STRING; t: test);
57  VAR
58    success: BOOLEAN;
59  BEGIN (* RunTest *)
60    t(success);
61    IF (success) THEN
62      BEGIN (* IF *)
63        WriteLn('PASSED - ', name)
64      END (* IF *)
65    ELSE
66      BEGIN (* ELSE *)
67        WriteLn('FAILED - ', name);
68        Halt(1);
69      END; (* ELSE *)
70  END; (* RunTest *)
71
72  BEGIN (* TestQADS *)
73    RunTest('InitialQueue_IsEmpty', InitialQueue_IsEmpty);
74    RunTest('Clear_EmptiesQueue', Clear_EmptiesQueue);
75    RunTest('Enqueue_AddsElement', Enqueue_AddsElement);
76    RunTest('DequeueEmptyQueue_ReturnsFalse',
      ↪  DequeueEmptyQueue_ReturnsFalse);
77
78    WriteLn('All tests passed');
79  END. (* TestQADS *)
```

3.4 Testergebnisse

```
elias@EliasLaptop:~/Repos/SEbaBB2/pascal/PascalWorkspace/UE4/bin$ ./TestQADS
PASSED - InitialQueue_IsEmpty
PASSED - Clear_EmptiesQueue
PASSED - Enqueue_AddsElement
PASSED - DequeueEmptyQueue_ReturnsFalse
All tests passed
```

Abbildung 3: Ausgabe des Testprogramms *TestQADS*