ADF2x & PRO2x **Übungen zu Fortgeschrittenen Algorithmen & Datenstrukturen und OOP** **SS 24, Übung 7**

Abgabetermin: Sa, 22.06.2024

☒ **Gr. 1,** S. Schöberl, MSc **Name** Elias Leonhardsberger **Aufwand in h** 7

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

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

1. „Behälter“ *Vector* als Klasse (8 Punkte)

Implementieren Sie eine Klasse *Vector*, die Werte vom Typ *INTEGER* aufnehmen kann. Ihre Klasse muss mindestens folgende Methoden zur Verfügung stellen:

`PROCEDURE` **`Add`**`(val: INTEGER);`
fügt den Wert *val* „hinten“ an.

`PROCEDURE` **`InsertElementAt`**`(pos: INTEGER; val: INTEGER; VAR ok: BOOLEAN);`
fügt den Wert *val* an der Stelle *pos* ein, wobei die Werte ab dieser Stelle nach hinten verschoben werden. Ist *pos* ≤ 0, wird *val* „vorne“ eingefügt; ist *pos* > *Size* (s. u.) wird *val* „hinten“ angefügt. Der Ausgangsparameter *ok* liefert nur dann *FALSE*, wenn für *pos* ein Wert über der Obergrenze des Vektors (*Capacity*) angegeben wurde.

`PROCEDURE` **`GetElementAt`**`(pos: INTEGER; VAR val: INTEGER; VAR ok: BOOLEAN);`
liefert den Wert *val* an der Stelle *pos*. Der Ausgangsparameter *ok* liefert nur dann *FALSE*, wenn für *pos* ein ungültiger Wert angegeben wurde.

`FUNCTION` **`Size`**`: INTEGER;`
liefert die aktuelle Anzahl der Elemente.

`PROCEDURE` **`Clear;`**
leert den Behälter (Size liefert dann 0).

Sie können die Größe des Vektors als vorgegeben (über Konstante) annehmen. Wenn Sie möchten, können Sie die Klasse *Vector* aber auch so implementieren, dass die Größe des Vektors erst beim Erstellen eines *Vector*-Objekts festgelegt wird. Implementieren Sie die folgende Methode entsprechend Ihrem Lösungsansatz.

`FUNCTION` **`Capacity`**`: INTEGER`
liefert die aktuelle Kapazität des Behälters (= max. Anzahl der Elemente, die der Vektor aufnehmen kann).

2. Komposition: Kellerspeicher (*stack*) und Warteschlange (*queue*) (4 + 4 Punkte)

Bauen Sie ohne den Einsatz von Vererbung und unter Verwendung des Vektors aus Aufgabe 1 zwei neue Behälter: einen Kellerspeicher (*stack*) und eine Warteschlange (*queue*).

a) Der Kellerspeicher *CardinalStack* lässt nur positive *INTEGER*-Werte als Elemente zu und bietet die für einen Kellerspeicher typischen Methoden *IsEmpty*, *Push* und *Pop*.

b) Die Warteschlange *EvenQueue* akzeptiert nur gerade *INTEGER*-Werte und muss mindestens die folgenden Methoden bieten: *IsEmpty*, *Enqueue* (Element einfügen) und *Dequeue* (Element entfernen).

3. Vererbung: Vektoren mit Einschränkung (3 + 5 Punkte)

Vektoren mit Einschränkung nehmen nur solche Werte auf, die einer Einschränkung genügen.

a) Leiten Sie von Ihrer Klasse *Vector* aus Aufgabe 1 eine neue Klasse *NaturalVector* ab, deren Objekte nur natürliche Zahlen aufnehmen. Überschreiben Sie dazu die beiden Methoden *Add* und *InsertElementAt*.

b) Leiten Sie von Ihrer Klasse *Vector* aus Aufgabe 1 eine Klasse *PrimeVector* ab, deren Objekte nur *Primzahlen* aufnehmen.

Testen Sie Ihre Klassen ausführlich und beschreiben Sie anhand der Testfälle, wo überall und auch warum Polymorphismus und dynamische Bindung zum Einsatz kommen bzw. kommen müssen.

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 UE07

Elias Leonhardsberger

21. Juni 2024, Hagenberg

Inhaltsverzeichnis

1 Vektor als Abstrakte Datenstruktur

1.1 Lösungsidee

Die Größe des Vektors wird statisch definiert, daher speichert der Vektor die Daten in einem einfachem Array.

RemoveElementAt wird zusätzlich hinzugefügt, um die Queue und den Stack zu ermöglichen. Hier müssen auch die Positionen der anderen Elemente angepasst werden.

Der Stack und die Queue sind Klassen die, mittels Komposition, den Vektor als Datenkomponente nutzen. Dadurch ist die Implementierung sehr einfach, da die Prozeduren eine einfache überprüfung und dann ein Vektor Aufruf sind.

Die Funktion isPrime wurde für den PrimeVector implementiert. Da sich die Multiplikationspaare ab der Wurzel einer Zahl wiederholen, muss man nur den Bereich von 2 bis zur Wurzel auf Teiler überprüfen.

Die Tests wurden wieder wie Unittests aufgebaut.

Die Tests für NaturalVector and PrimeVector werden bei den Vector Tests eingebaut. Die Tests werden mit einem Vector Pointer ausgeführt um zu zeigen, dass NaturalVector und PrimeVector mithilfe von Polymorphismus Vectors sind. Dynamische Bindung kommt zum Einsatz um mit einem *VectorPtr* das *Add* eines PrimeVector auszuführen.

Die Vector Tests nutzen nur positive Primzahlen, um zu zeigen, dass in diesem Fall die Vectoren gleich sind.

Die einzelnen Tests wurden mithilfe eines Skriptes ausgeführt und in ein File geschrieben.

```
./TestScript.sh &> ./TestResult.txt
```

1.2 Souce Code

1.2.1 UVector.pas

```
1
2  UNIT UVector;
3
4  INTERFACE
5
6  CONST
7    MAX_CAPACITY = 10;
8
9  TYPE
10   VectorPtr = ^Vector;
11   Vector = OBJECT
12     (* Initializes the vector. *)
13     CONSTRUCTOR Init;
14
15     (* Destroys the vector. *)
16     DESTRUCTOR Done; VIRTUAL;
17
18     (* Adds an element to the end of the vector. *)
```

```
19      (*  val - The value to be added to the vector. *)
20      (*  ok - A boolean value that will be set to TRUE if the operation
        ↪  was successful, FALSE otherwise. *)
21      PROCEDURE Add(val: INTEGER; VAR ok: BOOLEAN); VIRTUAL;
22
23      (* Inserts/sets the value of the element at the specified position,
        ↪  the other elements are moved accordinglly. *)
24      (* If the position is greater than the vSize of the vector, the
        ↪  element will be added at the end. *)
25      (* If the position is equal to or less than 0, the element will be
        ↪  added at the start. *)
26      (* If the position is greater than the capacity of the vector, the
        ↪  operation will fail. *)
27      (*  pos - The position of the element to be set. *)
28      (*  val - The value to be set. *)
29      (*  ok - A boolean value that will be set to TRUE if the operation
        ↪  was successful, FALSE otherwise. *)
30      PROCEDURE InsertElementAt(pos, val: INTEGER; VAR ok: BOOLEAN);
        ↪  VIRTUAL;
31
32      (* Gets the value of the element at the specified position. If the
        ↪  position is greater than the vSize of the vector, the operation
        ↪  will fail. *)
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 GetElementAt(pos: INTEGER; VAR val: INTEGER; VAR ok:
        ↪  BOOLEAN);
37
38      (* Removes the element at the specified position, the other elements
        ↪  are moved accordinglly. If the position is greater than the
        ↪  vSize of the vector, the operation will fail. *)
39      (*  pos - The position of the element to be retrieved. *)
40      (*  val - The value of the element at the specified position. *)
41      (*  ok - A boolean value that will be set to TRUE if the operation
        ↪  was successful, FALSE otherwise. *)
42      PROCEDURE RemoveElementAt(pos: INTEGER; VAR ok: BOOLEAN);
43
44      (* Returns the number of elements in the vector. *)
45      FUNCTION Size: INTEGER;
46
47      (* Returns the current capacity of the vector. *)
48      FUNCTION Capacity: INTEGER;
49
50      (* Clears the vector and sets its capacity to the base value. *)
51      PROCEDURE Clear;
```

```
52
53      PRIVATE
54        data: ARRAY[1..MAX_CAPACITY] OF INTEGER;
55        vSize: INTEGER;
56    END;
57
58  IMPLEMENTATION
59
60  CONSTRUCTOR Vector.Init;
61  BEGIN (* Init *)
62    vSize := 0;
63  END; (* Init *)
64
65  DESTRUCTOR Vector.Done;
66  BEGIN (* Done *)
67    (* Nothing to do here, because the array is static. *)
68  END; (* Done *)
69
70  PROCEDURE Vector.Add(val: INTEGER; VAR ok: BOOLEAN);
71  BEGIN
72    IF (vSize >= MAX_CAPACITY) THEN
73      BEGIN (* IF *)
74        ok := FALSE;
75      END (* IF *)
76    ELSE
77      BEGIN (* ELSE *)
78        ok := TRUE;
79        vSize := vSize + 1;
80        data[vSize] := val;
81      END; (* ELSE *)
82  END; (* Add *)
83
84  PROCEDURE Vector.InsertElementAt(pos, val: INTEGER; VAR ok: BOOLEAN);
85  VAR
86    i: INTEGER;
87  BEGIN (* InsertElementAt *)
88    IF (pos > MAX_CAPACITY) OR (vSize + 1 > Capacity) THEN
89      BEGIN (* IF *)
90        ok := FALSE;
91      END (* IF *)
92    ELSE
93      IF (pos > vSize) THEN
94        BEGIN (* ELSE IF *)
95          Add(val, ok);
96        END (* ELSE IF *)
97    ELSE
98      BEGIN (* ELSE *)
```

```
99        IF (pos <= 0) THEN
100         BEGIN (* ELSE IF *)
101           pos := 1;
102         END; (* ELSE IF *)
103
104       vSize := vSize + 1;
105
106       FOR i := vSize DOWNTO pos + 1 DO
107         BEGIN (* FOR *)
108           data[i] := data[i-1];
109         END; (* FOR *)
110
111       data[pos] := val;
112       ok := TRUE;
113     END; (* ELSE *)
114 END; (* InsertElementAt *)
115
116 PROCEDURE Vector.GetElementAt(pos: INTEGER; VAR val: INTEGER; VAR ok:
    ↪  BOOLEAN);
117 BEGIN (* GetElementAt *)
118   IF (pos > vSize) OR (pos < 1) THEN
119     BEGIN (* IF *)
120       ok := FALSE;
121     END (* IF *)
122   ELSE
123     BEGIN (* ELSE *)
124       val := data[pos];
125       ok := TRUE;
126     END; (* ELSE *)
127 END; (* GetElementAt *)
128
129 PROCEDURE Vector.RemoveElementAt(pos: INTEGER; VAR ok: BOOLEAN);
130 VAR
131   i: INTEGER;
132 BEGIN (* RemoveElementAt *)
133   IF (pos > vSize) OR (pos < 1) THEN
134     BEGIN (* IF *)
135       ok := FALSE;
136     END (* IF *)
137   ELSE
138     BEGIN (* ELSE *)
139       vSize := vSize - 1;
140       ok := TRUE;
141
142       FOR i := pos TO vSize DO
143         BEGIN (* FOR *)
144           data[i] := data[i+1];
```

```
145         END; (* FOR *)
146     END; (* ELSE *)
147 END; (* RemoveElementAt *)
148 
149 FUNCTION Vector.Size: INTEGER;
150 BEGIN (* Size *)
151   Size := vSize;
152 END; (* Size *)
153 
154 FUNCTION Vector.Capacity: INTEGER;
155 BEGIN (* Capacity *)
156   Capacity := MAX_CAPACITY;
157 END; (* Capacity *)
158 
159 PROCEDURE Vector.Clear;
160 BEGIN (* Clear *)
161   vSize := 0;
162 END; (* Clear *)
163 
164 END.
```

1.2.2 UCardinalStack.pas

```
1  
2  UNIT UCardinalStack;
3  
4  INTERFACE
5  
6  USES
7  UVector;
8  
9  TYPE
10   CardinalStack = OBJECT
11 
12     (* Initializes the Stack *)
13     CONSTRUCTOR Init;
14 
15     (* Destroys the Stack *)
16     DESTRUCTOR Done;
17 
18     (* Pushes a value onto the Stack *)
19     (*  val - The value to be pushed to the stack *)
20     (*  ok - A boolean value that is set to TRUE if the push was
       ↪  successful, FALSE otherwise *)
21     PROCEDURE Push(val: INTEGER; VAR ok: BOOLEAN);
22 
23     (* Pops a value from the Stack *)
24     (*  val - The value that was popped from the stack *)
```

```
25      (*  ok - A boolean value that is set to TRUE if the pop was
        ↪  successful, FALSE otherwise *)
26      PROCEDURE Pop(VAR val: INTEGER; VAR ok: BOOLEAN);
27
28      (* Checks if the Stack is empty *)
29      FUNCTION IsEmpty: BOOLEAN;
30
31      PRIVATE
32        data: Vector;
33
34    END;
35
36  IMPLEMENTATION
37
38  CONSTRUCTOR CardinalStack.Init;
39  BEGIN (* Init *)
40    data.Init();
41  END; (* Init *)
42
43  DESTRUCTOR CardinalStack.Done;
44  BEGIN (* Done *)
45    data.Done();
46  END; (* Done *)
47
48  PROCEDURE CardinalStack.Push(val: INTEGER; VAR ok: BOOLEAN);
49  BEGIN (* Push *)
50    IF (val >= 0) THEN
51      BEGIN (* IF *)
52        data.InsertElementAt(0, val, ok);
53      END (* IF *)
54    ELSE
55      BEGIN (* ELSE *)
56        ok := FALSE;
57      END; (* ELSE *)
58  END; (* Push *)
59
60  PROCEDURE CardinalStack.Pop(VAR val: INTEGER; VAR ok: BOOLEAN);
61  BEGIN (* Pop *)
62    data.GetElementAt(1, val, ok);
63
64    IF ok THEN
65      BEGIN (* IF *)
66        data.RemoveElementAt(1, ok);
67      END (* IF *)
68  END; (* Pop *)
69
70  FUNCTION CardinalStack.IsEmpty: BOOLEAN;
```

```
71  BEGIN (* IsEmpty *)
72    IsEmpty := data.Size() = 0;
73  END; (* IsEmpty *)
74
75  END.
```

1.2.3 UEvenQueue.pas

```
1
2   UNIT UEvenQueue;
3
4   INTERFACE
5
6   USES
7   UVector;
8
9   TYPE
10    EvenQueue = OBJECT
11
12      (* Initializes the Queue *)
13      CONSTRUCTOR Init;
14
15      (* Destroys the Queue *)
16      DESTRUCTOR Done;
17
18      (* Enqueues a value into the Queue *)
19      (*  val - The value to be euqueued to the queue *)
20      (*  ok - A boolean value that is set to TRUE if the enqueue was
        ↪  successful, FALSE otherwise *)
21      PROCEDURE Enqueue(val: INTEGER; VAR ok: BOOLEAN);
22
23      (* Dequeues a value from the Queue *)
24      (*  val - The value that was dequeued from the queue *)
25      (*  ok - A boolean value that is set to TRUE if the dequeue was
        ↪  successful, FALSE otherwise *)
26      PROCEDURE Dequeue(VAR val: INTEGER; VAR ok: BOOLEAN);
27
28      (* Checks if the Queue is empty *)
29      FUNCTION IsEmpty: BOOLEAN;
30
31      PRIVATE
32        data: Vector;
33
34    END;
35
36  IMPLEMENTATION
37
38  CONSTRUCTOR EvenQueue.Init;
```

```
39 BEGIN (* Init *)
40   data.Init();
41 END; (* Init *)
42
43 DESTRUCTOR EvenQueue.Done;
44 BEGIN (* Done *)
45   data.Done();
46 END; (* Done *)
47
48 PROCEDURE EvenQueue.Enqueue(val: INTEGER; VAR ok: BOOLEAN);
49 BEGIN (* Enqueue *)
50   IF val MOD 2 = 0 THEN
51     BEGIN (* IF *)
52       data.Add(val, ok);
53     END (* IF *)
54   ELSE
55     BEGIN (* ELSE *)
56       ok := FALSE;
57     END; (* ELSE *)
58 END; (* Enqueue *)
59
60 PROCEDURE EvenQueue.Dequeue(VAR val: INTEGER; VAR ok: BOOLEAN);
61 BEGIN (* Dequeue *)
62   data.GetElementAt(1, val, ok);
63
64   IF ok THEN
65     BEGIN (* IF *)
66       data.RemoveElementAt(1, ok);
67     END (* IF *)
68 END; (* Dequeue *)
69
70 FUNCTION EvenQueue.IsEmpty: BOOLEAN;
71 BEGIN (* IsEmpty *)
72   IsEmpty := data.Size() = 0;
73 END; (* IsEmpty *)
74
75 END.
```

1.2.4 UNaturalVector.pas

```
1
2 UNIT UNaturalVector;
3
4 INTERFACE
5
6 USES
7 UVector;
8
```

```
9
10  TYPE
11    NaturalVectorPtr = ^NaturalVector;
12    NaturalVector = OBJECT(Vector)
13      (* Initializes the vector. *)
14      CONSTRUCTOR Init;
15
16      (* Destroys the vector. *)
17      DESTRUCTOR Done; VIRTUAL;
18
19      (* Adds an element to the end of the vector. *)
20      (* If the value is less than 0, the operation will fail. *)
21      (*  val - The value to be added to the vector. *)
22      (*  ok - A boolean value that will be set to TRUE if the operation
        ↪  was successful, FALSE otherwise. *)
23      PROCEDURE Add(val: INTEGER; VAR ok: BOOLEAN); VIRTUAL;
24
25      (* Inserts/sets the value of the element at the specified position,
        ↪  the other elements are moved accordinglly. *)
26      (* If the value is less than 0, the operation will fail. *)
27      (* If the position is greater than the vSize of the vector, the
        ↪  element will be added at the end. *)
28      (* If the position is equal to or less than 0, the element will be
        ↪  added at the start. *)
29      (* If the position is greater than the capacity of the vector, the
        ↪  operation will fail. *)
30      (*  pos - The position of the element to be set. *)
31      (*  val - The value to be set. *)
32      (*  ok - A boolean value that will be set to TRUE if the operation
        ↪  was successful, FALSE otherwise. *)
33      PROCEDURE InsertElementAt(pos, val: INTEGER; VAR ok: BOOLEAN);
        ↪  VIRTUAL;
34    END;
35
36  IMPLEMENTATION
37
38  CONSTRUCTOR NaturalVector.Init;
39  BEGIN (* Init *)
40    INHERITED Init();
41  END; (* Init *)
42
43  DESTRUCTOR NaturalVector.Done;
44  BEGIN (* Done *)
45    INHERITED Done();
46  END; (* Done *)
47
48  PROCEDURE NaturalVector.Add(val: INTEGER; VAR ok: BOOLEAN);
```

```
49  BEGIN (* Add *)
50    IF (val >= 0) THEN
51      BEGIN (* IF *)
52        INHERITED Add(val, ok);
53      END (* IF *)
54    ELSE
55      BEGIN (* ELSE *)
56        ok := FALSE;
57      END; (* ELSE *)
58  END; (* Add *)
59
60  PROCEDURE NaturalVector.InsertElementAt(pos, val: INTEGER; VAR ok:
    ↪  BOOLEAN);
61  BEGIN (* InsertElementAt *)
62    IF (val >= 0) THEN
63      BEGIN (* IF *)
64        INHERITED InsertElementAt(pos, val, ok);
65      END (* IF *)
66    ELSE
67      BEGIN (* ELSE *)
68        ok := FALSE;
69      END; (* ELSE *)
70  END; (* InsertElementAt *)
71
72  END.
```

1.2.5 UPrimeVector.pas

```
1
2   UNIT UPrimeVector;
3
4   INTERFACE
5
6   USES
7   UVector;
8
9
10  TYPE
11    PrimeVectorPtr = ^PrimeVector;
12    PrimeVector = OBJECT(Vector)
13      (* Initializes the vector. *)
14      CONSTRUCTOR Init;
15
16      (* Destroys the vector. *)
17      DESTRUCTOR Done; VIRTUAL;
18
19      (* Adds an element to the end of the vector. *)
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(val: INTEGER; VAR ok: BOOLEAN); VIRTUAL;
23
24      (* Inserts/sets the value of the element at the specified position,
        ↪  the other elements are moved accordinglly. *)
25      (* If the position is greater than the vSize of the vector, the
        ↪  element will be added at the end. *)
26      (* If the position is equal to or less than 0, the element will be
        ↪  added at the start. *)
27      (* If the position is greater than the capacity of the vector, the
        ↪  operation will fail. *)
28      (*  pos - The position of the element to be set. *)
29      (*  val - The value to be set. *)
30      (*  ok - A boolean value that will be set to TRUE if the operation
        ↪  was successful, FALSE otherwise. *)
31      PROCEDURE InsertElementAt(pos, val: INTEGER; VAR ok: BOOLEAN);
        ↪  VIRTUAL;
32    END;
33
34  IMPLEMENTATION
35
36  CONSTRUCTOR PrimeVector.Init;
37  BEGIN (* Init *)
38    INHERITED Init();
39  END; (* Init *)
40
41  DESTRUCTOR PrimeVector.Done;
42  BEGIN (* Done *)
43    INHERITED Done();
44  END; (* Done *)
45
46  FUNCTION IsPrime(n: INTEGER): BOOLEAN;
47  VAR
48    i: INTEGER;
49    result: BOOLEAN;
50  BEGIN (* IsPrime *)
51    result := TRUE;
52    n := Abs(n);
53
54    IF (n <= 1) THEN
55      BEGIN (* IF *)
56        result := FALSE;
57      END (* IF *)
58    ELSE
59      BEGIN (* ELSE *)
60        i := 2;
```

```
61        WHILE (i <= Round(Sqrt(n))) AND (result) DO
62          BEGIN (* WHILE *)
63            IF (n MOD i = 0) THEN
64              BEGIN (* IF *)
65                result := FALSE;
66              END; (* IF *)
67            i := i + 1;
68          END; (* WHILE *)
69      END; (* ELSE *)
70
71    IsPrime := result;
72  END; (* IsPrime *)
73
74  PROCEDURE PrimeVector.Add(val: INTEGER; VAR ok: BOOLEAN);
75  BEGIN (* Add *)
76    IF (IsPrime(val)) THEN
77      BEGIN (* IF *)
78        INHERITED Add(val, ok);
79      END (* IF *)
80    ELSE
81      BEGIN (* ELSE *)
82        ok := FALSE;
83      END; (* ELSE *)
84  END; (* Add *)
85
86  PROCEDURE PrimeVector.InsertElementAt(pos, val: INTEGER; VAR ok:
    ↪  BOOLEAN);
87  BEGIN (* InsertElementAt *)
88    IF (IsPrime(val)) THEN
89      BEGIN (* IF *)
90        INHERITED InsertElementAt(pos, val, ok);
91      END (* IF *)
92    ELSE
93      BEGIN (* ELSE *)
94        ok := FALSE;
95      END; (* ELSE *)
96  END; (* InsertElementAt *)
97
98  END.
```

1.3 Tests

1.3.1 TestVector.pas

```
1  PROGRAM TestVector;
2
3  USES
4  UVector, UNaturalVector, UPrimeVector;
5
6  CONST
7    PrimeArray: ARRAY [0..11] OF INTEGER = (2, 3, 5, 7, 11, 13, 17, 19, 23,
     ↪  29, 31, 37);
8
9  TYPE
10   test = PROCEDURE (v: VectorPtr; VAR success: BOOLEAN);
11
12 PROCEDURE InitialVector_IsEmpty(v: VectorPtr; VAR success: BOOLEAN);
13 BEGIN (* InitialVector_IsEmpty *)
14   success := (v^.Size() = 0)
15              AND (v^.Capacity() = 10);
16 END; (* InitialVector_IsEmpty *)
17
18 PROCEDURE ClearVector_IsEmpty(v: VectorPtr; VAR success: BOOLEAN);
19 VAR
20   addOk: BOOLEAN;
21 BEGIN (* ClearVector_IsEmpty *)
22   v^.Add(PrimeArray[1], addOk);
23   v^.Clear();
24   success := addOk
25              AND (v^.Size() = 0);
26 END; (* ClearVector_IsEmpty *)
27
28 PROCEDURE AddElement_IncreasesSize(v: VectorPtr; VAR success: BOOLEAN);
29 VAR
30   addOk: BOOLEAN;
31 BEGIN (* AddElement_IncreasesSize *)
32   v^.Add(PrimeArray[1], addOk);
33   success := addOk
34              AND (v^.Size() = 1);
35 END; (* AddElement_IncreasesSize *)
36
37 PROCEDURE AddElementsOverCapacity_OkFalse(v: VectorPtr; VAR success:
     ↪  BOOLEAN);
38 VAR
39   addOk: BOOLEAN;
40   i: INTEGER;
41 BEGIN (* AddElementsOverCapacity_OkFalse *)
42   success := TRUE;
```

```
43
44    FOR i := 1 TO 10 DO
45      BEGIN (* FOR *)
46        v^.Add(PrimeArray[i], addOk);
47        success := success AND addOk;
48      END; (* FOR *)
49
50    v^.Add(PrimeArray[11], addOk);
51
52    success := success
53               AND NOT addOk
54               AND (v^.Size() = 10);
55  END; (* AddElementsOverCapacity_OkFalse *)
56
57  PROCEDURE GetElementAt_ReturnsCorrectElement(v: VectorPtr; VAR success:
    ↪  BOOLEAN);
58  VAR
59    addOk, elementAtOk: BOOLEAN;
60    i, element: INTEGER;
61  BEGIN (* GetElementAt_ReturnsCorrectElement *)
62    success := TRUE;
63
64    FOR i := 1 TO 10 DO
65      BEGIN (* FOR *)
66        v^.Add(PrimeArray[i], addOk);
67        success := success
68                   AND addOk
69                   AND (v^.Size() = i);
70      END; (* FOR *)
71
72    FOR i := 1 TO 10 DO
73      BEGIN (* FOR *)
74        v^.GetElementAt(i, element, elementAtOk);
75        success := success
76                   AND elementAtOk
77                   AND (element = PrimeArray[i]);
78      END; (* FOR *)
79  END; (* GetElementAt_ReturnsCorrectElement *)
80
81  PROCEDURE GetElementAtOutOfBounds_OkFalse(v: VectorPtr; VAR success:
    ↪  BOOLEAN);
82  VAR
83    addOk, elementAtOk: BOOLEAN;
84    i, element: INTEGER;
85  BEGIN (* GetElementAtOutOfBounds_OkFalse *)
86    success := TRUE;
87
```

```
88    FOR i := 1 TO 5 DO
89      BEGIN (* FOR *)
90        v^.Add(PrimeArray[i], addOk);
91        success := success AND addOk;
92      END; (* FOR *)
93
94    v^.GetElementAt(6, element, elementAtOk);
95    success := success
96               AND NOT elementAtOk;
97
98    v^.GetElementAt(0, element, elementAtOk);
99    success := success
100              AND NOT elementAtOk
101              AND (v^.Size() = 5);
102 END; (* GetElementAtOutOfBounds_OkFalse *)
103
104 PROCEDURE InsertElementAt_AddsElementToPosition(v: VectorPtr; VAR
    ↪  success: BOOLEAN);
105 VAR
106   addOk, insertOk, elementAtOk: BOOLEAN;
107   i, element: INTEGER;
108 BEGIN (* InsertElementAt_AddsElementToPosition *)
109   success := TRUE;
110
111   FOR i := 1 TO 9 DO
112     BEGIN (* FOR *)
113       v^.Add(PrimeArray[i], addOk);
114       success := success AND addOk;
115     END; (* FOR *)
116
117   v^.InsertElementAt(5, PrimeArray[11], insertOk);
118   v^.GetElementAt(5, element, elementAtOk);
119   success := success
120              AND insertOk
121              AND elementAtOk
122              AND (element = PrimeArray[11])
123              AND (v^.Size() = 10);
124
125   FOR i := 6 TO 10 DO
126     BEGIN (* FOR *)
127       v^.GetElementAt(i, element, elementAtOk);
128       success := success
129                  AND elementAtOk
130                  AND (element = PrimeArray[i - 1]);
131     END; (* FOR *)
132 END; (* InsertElementAt_AddsElementToPosition *)
133
```

```
134 PROCEDURE InsertElementAtGreaterThanCapacity_DoesNotInsert(v: VectorPtr;
    ↪  VAR success: BOOLEAN);
135 VAR
136   insertOk: BOOLEAN;
137 BEGIN (* InsertElementAtGreaterThanCapacity_DoesNotInsert *)
138   v^.InsertElementAt(11, PrimeArray[1], insertOk);
139   success := NOT insertOk
140              AND (v^.Size() = 0);
141 END; (* InsertElementAtGreaterThanCapacity_DoesNotInsert *)
142
143 PROCEDURE InsertElementAtLessThanOne_AddsToFront(v: VectorPtr; VAR
    ↪  success: BOOLEAN);
144 VAR
145   addOk, insertOk, elementAtOk: BOOLEAN;
146   i, element: INTEGER;
147 BEGIN (* InsertElementAtLessThanOne_AddsToFront *)
148   success := TRUE;
149
150   FOR i := 1 TO 9 DO
151     BEGIN (* FOR *)
152       v^.Add(PrimeArray[i], addOk);
153       success := success AND addOk;
154     END; (* FOR *)
155
156   v^.InsertElementAt(-17, PrimeArray[11], insertOk);
157   v^.GetElementAt(1, element, elementAtOk);
158   success := success
159              AND insertOk
160              AND elementAtOk
161              AND (element = PrimeArray[11])
162              AND (v^.Size() = 10);
163
164   FOR i := 2 TO 10 DO
165     BEGIN (* FOR *)
166       v^.GetElementAt(i, element, elementAtOk);
167       success := success
168                  AND elementAtOk
169                  AND (element = PrimeArray[i - 1]);
170     END; (* FOR *)
171 END; (* InsertElementAtLessThanOne_AddsToFront *)
172
173 PROCEDURE InsertElementAtGreaterThanSize_AddsToEnd(v: VectorPtr; VAR
    ↪  success: BOOLEAN);
174 VAR
175   addOk, insertOk, elementAtOk: BOOLEAN;
176   i, element: INTEGER;
177 BEGIN (* InsertElementAtGreaterThanSize_AddsToEnd *)
```

```
178   success := TRUE;
179
180   FOR i := 1 TO 5 DO
181     BEGIN (* FOR *)
182       v^.Add(PrimeArray[i], addOk);
183       success := success AND addOk;
184     END; (* FOR *)
185
186   v^.InsertElementAt(10, PrimeArray[11], insertOk);
187   v^.GetElementAt(6, element, elementAtOk);
188   success := success
189              AND insertOk
190              AND elementAtOk
191              AND (element = PrimeArray[11])
192              AND (v^.Size() = 6);
193
194   FOR i := 1 TO 5 DO
195     BEGIN (* FOR *)
196       v^.GetElementAt(i, element, elementAtOk);
197       success := success AND elementAtOk AND (element = PrimeArray[i]);
198     END; (* FOR *)
199 END; (* InsertElementAtGreaterThanSize_AddsToEnd *)
200
201 PROCEDURE InsertElementAtAlreadyFull_OkFalse(v: VectorPtr; VAR success:
    ↪  BOOLEAN);
202 VAR
203   addOk, insertOk, elementAtOk: BOOLEAN;
204   i, element: INTEGER;
205 BEGIN (* InsertElementAtAlreadyFull_OkFalse *)
206   success := TRUE;
207
208   FOR i := 1 TO 10 DO
209     BEGIN (* FOR *)
210       v^.Add(PrimeArray[i], addOk);
211       success := success AND addOk;
212     END; (* FOR *)
213
214   v^.InsertElementAt(5, PrimeArray[11], insertOk);
215   success := success
216              AND NOT insertOk
217              AND (v^.Size() = 10);
218
219   FOR i := 1 TO 10 DO
220     BEGIN (* FOR *)
221       v^.GetElementAt(i, element, elementAtOk);
222       success := success
223                  AND elementAtOk
```

```
224                   AND (element = PrimeArray[i]);
225     END; (* FOR *)
226 END; (* InsertElementAtAlreadyFull_OkFalse *)
227
228 PROCEDURE RemoveElementAt_RemovesElement(v: VectorPtr; VAR success:
    ↪  BOOLEAN);
229 VAR
230   addOk, removeOk, elementAtOk: BOOLEAN;
231   i, element: INTEGER;
232 BEGIN (* RemoveElementAt_RemovesElement *)
233   success := TRUE;
234
235   FOR i := 1 TO 10 DO
236     BEGIN (* FOR *)
237       v^.Add(PrimeArray[i], addOk);
238       success := success AND addOk;
239     END; (* FOR *)
240
241   v^.RemoveElementAt(5, removeOk);
242   success := success
243              AND (v^.Size() = 9);
244
245   FOR i := 1 TO 4 DO
246     BEGIN (* FOR *)
247       v^.GetElementAt(i, element, elementAtOk);
248       success := success AND elementAtOk AND (element = PrimeArray[i]);
249     END; (* FOR *)
250
251   FOR i := 5 TO 9 DO
252     BEGIN (* FOR *)
253       v^.GetElementAt(i, element, elementAtOk);
254       success := success AND elementAtOk AND (element = PrimeArray[i +
        ↪  1]);
255     END; (* FOR *)
256 END; (* RemoveElementAt_RemovesElement *)
257
258 PROCEDURE RemoveElementAtOutOfBounds_OkFalse(v: VectorPtr; VAR success:
    ↪  BOOLEAN);
259 VAR
260   removeOk, addOk: BOOLEAN;
261   i: INTEGER;
262 BEGIN (* RemoveElementAtOutOfBounds_OkFalse *)
263   success := TRUE;
264
265   FOR i := 1 TO 5 DO
266     BEGIN (* FOR *)
267       v^.Add(PrimeArray[i], addOk);
```

```
268         success := success AND addOk;
269       END; (* FOR *)
270
271     v^.RemoveElementAt(6, removeOk);
272     success := success AND NOT removeOk;
273     v^.RemoveElementAt(0, removeOk);
274     success := success
275                AND NOT removeOk
276                AND (v^.Size() = 5);
277   END; (* RemoveElementAtOutOfBounds_OkFalse *)
278
279   PROCEDURE NonPrime_AddsAndInserts(v: VectorPtr; VAR success: BOOLEAN);
280   VAR
281     addOk, insertOk: BOOLEAN;
282   BEGIN (* NonPrime_AddsAndInserts *)
283     v^.Add(4, addOk);
284     v^.InsertElementAt(0, 6, insertOk);
285
286     success := addOk
287                AND insertOk
288                AND (v^.Size() = 2);
289   END; (* NonPrime_AddsAndInserts *)
290
291   PROCEDURE NonPrime_DoesNotAddAndInsert(v: VectorPtr; VAR success:
      ↪  BOOLEAN);
292   VAR
293     addOk, insertOk: BOOLEAN;
294   BEGIN (* NonPrime_DoesNotAddAndInsert *)
295     v^.Add(4, addOk);
296     v^.InsertElementAt(0, 6, insertOk);
297
298     success := NOT addOk
299                AND NOT insertOk
300                AND (v^.Size() = 0);
301   END; (* NonPrime_DoesNotAddAndInsert *)
302
303   PROCEDURE NegativeNumber_AddsAndInserts(v: VectorPtr; VAR success:
      ↪  BOOLEAN);
304   VAR
305     addOk, insertOk: BOOLEAN;
306   BEGIN (* NegativeNumber_AddsAndInserts *)
307     v^.Add(-2, addOk);
308     v^.InsertElementAt(0, -3, insertOk);
309
310     success := addOk
311                AND insertOk
312                AND (v^.Size() = 2);
```

```
313 END; (* NegativeNumber_AddsAndInserts *)
314
315 PROCEDURE NegativeNumber_DoesNotAddAndInsert(v: VectorPtr; VAR success:
    ↪  BOOLEAN);
316 VAR
317   addOk, insertOk: BOOLEAN;
318 BEGIN (* NegativeNumber_DoesNotAddAndInsert *)
319   v^.Add(-2, addOk);
320   v^.InsertElementAt(0, -3, insertOk);
321
322   success := NOT addOk
323              AND NOT insertOk
324              AND (v^.Size() = 0);
325 END; (* NegativeNumber_DoesNotAddAndInsert *)
326
327 PROCEDURE RunVectorTest(NAME: STRING; t: test);
328 VAR
329   success: BOOLEAN;
330   v: VectorPtr;
331 BEGIN (* RunVectorTest *)
332   NEW(v, Init());
333   t(v, success);
334   DISPOSE(v, Done());
335
336   IF (success) THEN
337     BEGIN (* IF *)
338       WriteLn('PASSED - ', name);
339     END (* IF *)
340   ELSE
341     BEGIN (* ELSE *)
342       WriteLn('FAILED - ', name);
343       Halt(1);
344     END; (* ELSE *)
345 END; (* RunVectorTest *)
346
347 PROCEDURE RunNaturalVectorTest(NAME: STRING; t: test);
348 VAR
349   success: BOOLEAN;
350   v: VectorPtr;
351 BEGIN (* RunNaturalVectorTest *)
352   v := NEW(NaturalVectorPtr, Init());
353   t(v, success);
354   DISPOSE(v, Done());
355
356   IF (success) THEN
357     BEGIN (* IF *)
358       WriteLn('PASSED - ', name);
```

```
359       END (* IF *)
360     ELSE
361       BEGIN (* ELSE *)
362         WriteLn('FAILED - ', name);
363         Halt(1);
364       END; (* ELSE *)
365   END; (* RunNaturalVectorTest *)
366
367   PROCEDURE RunPrimeVectorTest(NAME: STRING; t: test);
368   VAR
369     success: BOOLEAN;
370     v: VectorPtr;
371   BEGIN (* RunPrimeVectorTest *)
372     v := NEW(PrimeVectorPtr, Init());
373     t(v, success);
374     DISPOSE(v, Done());
375
376     IF (success) THEN
377       BEGIN (* IF *)
378         WriteLn('PASSED - ', name);
379       END (* IF *)
380     ELSE
381       BEGIN (* ELSE *)
382         WriteLn('FAILED - ', name);
383         Halt(1);
384       END; (* ELSE *)
385   END; (* RunPrimeVectorTest *)
386
387   BEGIN (* TestVector *)
388     WriteLn('Vector:');
389     RunVectorTest('InitialVector_IsEmpty', InitialVector_IsEmpty);
390     RunVectorTest('ClearVector_IsEmpty', ClearVector_IsEmpty);
391     RunVectorTest('AddElement_IncreasesSize', AddElement_IncreasesSize);
392     RunVectorTest('AddElementsOverCapacity_OkFalse',
        ↪ AddElementsOverCapacity_OkFalse);
393     RunVectorTest('GetElementAt_ReturnsCorrectElement',
        ↪ GetElementAt_ReturnsCorrectElement);
394     RunVectorTest('GetElementAtOutOfBounds_OkFalse',
        ↪ GetElementAtOutOfBounds_OkFalse);
395     RunVectorTest('InsertElementAt_AddsElementToPosition',
        ↪ InsertElementAt_AddsElementToPosition);
396     RunVectorTest('InsertElementAtGreaterThanCapacity_DoesNotInsert',
        ↪ InsertElementAtGreaterThanCapacity_DoesNotInsert);
397     RunVectorTest('InsertElementAtLessThanOne_AddsToFront',
        ↪ InsertElementAtLessThanOne_AddsToFront);
398     RunVectorTest('InsertElementAtGreaterThanSize_AddsToEnd',
        ↪ InsertElementAtGreaterThanSize_AddsToEnd);
```

```
399     RunVectorTest('InsertElementAtAlreadyFull_OkFalse',
        ↪ InsertElementAtAlreadyFull_OkFalse);
400     RunVectorTest('RemoveElementAt_RemovesElement',
        ↪ RemoveElementAt_RemovesElement);
401     RunVectorTest('RemoveElementAtOutOfBounds_OkFalse',
        ↪ RemoveElementAtOutOfBounds_OkFalse);
402     RunVectorTest('NonPrime_AddsAndInserts', NonPrime_AddsAndInserts);
403     RunVectorTest('NegativeNumber_AddsAndInserts',
        ↪ NegativeNumber_AddsAndInserts);
404
405     WriteLn();
406     WriteLn('NaturalVector:');
407     RunNaturalVectorTest('InitialVector_IsEmpty', InitialVector_IsEmpty);
408     RunNaturalVectorTest('ClearVector_IsEmpty', ClearVector_IsEmpty);
409     RunNaturalVectorTest('AddElement_IncreasesSize',
        ↪ AddElement_IncreasesSize);
410     RunNaturalVectorTest('AddElementsOverCapacity_OkFalse',
        ↪ AddElementsOverCapacity_OkFalse);
411     RunNaturalVectorTest('GetElementAt_ReturnsCorrectElement',
        ↪ GetElementAt_ReturnsCorrectElement);
412     RunNaturalVectorTest('GetElementAtOutOfBounds_OkFalse',
        ↪ GetElementAtOutOfBounds_OkFalse);
413     RunNaturalVectorTest('InsertElementAt_AddsElementToPosition',
        ↪ InsertElementAt_AddsElementToPosition);
414
        ↪ RunNaturalVectorTest('InsertElementAtGreaterThanCapacity_DoesNotInsert',
        ↪ InsertElementAtGreaterThanCapacity_DoesNotInsert);
415     RunNaturalVectorTest('InsertElementAtLessThanOne_AddsToFront',
        ↪ InsertElementAtLessThanOne_AddsToFront);
416     RunNaturalVectorTest('InsertElementAtGreaterThanSize_AddsToEnd',
        ↪ InsertElementAtGreaterThanSize_AddsToEnd);
417     RunNaturalVectorTest('InsertElementAtAlreadyFull_OkFalse',
        ↪ InsertElementAtAlreadyFull_OkFalse);
418     RunNaturalVectorTest('RemoveElementAt_RemovesElement',
        ↪ RemoveElementAt_RemovesElement);
419     RunNaturalVectorTest('RemoveElementAtOutOfBounds_OkFalse',
        ↪ RemoveElementAtOutOfBounds_OkFalse);
420     RunNaturalVectorTest('NonPrime_AddsAndInserts',
        ↪ NonPrime_AddsAndInserts);
421     RunNaturalVectorTest('NegativeNumber_DoesNotAddAndInsert',
        ↪ NegativeNumber_DoesNotAddAndInsert);
422
423     WriteLn();
424     WriteLn('PrimeVector:');
425     RunPrimeVectorTest('InitialVector_IsEmpty', InitialVector_IsEmpty);
426     RunPrimeVectorTest('ClearVector_IsEmpty', ClearVector_IsEmpty);
```

```
427   RunPrimeVectorTest('AddElement_IncreasesSize',
      ↪  AddElement_IncreasesSize);
428   RunPrimeVectorTest('AddElementsOverCapacity_OkFalse',
      ↪  AddElementsOverCapacity_OkFalse);
429   RunPrimeVectorTest('GetElementAt_ReturnsCorrectElement',
      ↪  GetElementAt_ReturnsCorrectElement);
430   RunPrimeVectorTest('GetElementAtOutOfBounds_OkFalse',
      ↪  GetElementAtOutOfBounds_OkFalse);
431   RunPrimeVectorTest('InsertElementAt_AddsElementToPosition',
      ↪  InsertElementAt_AddsElementToPosition);
432   RunPrimeVectorTest('InsertElementAtGreaterThanCapacity_DoesNotInsert',
      ↪  InsertElementAtGreaterThanCapacity_DoesNotInsert);
433   RunPrimeVectorTest('InsertElementAtLessThanOne_AddsToFront',
      ↪  InsertElementAtLessThanOne_AddsToFront);
434   RunPrimeVectorTest('InsertElementAtGreaterThanSize_AddsToEnd',
      ↪  InsertElementAtGreaterThanSize_AddsToEnd);
435   RunPrimeVectorTest('InsertElementAtAlreadyFull_OkFalse',
      ↪  InsertElementAtAlreadyFull_OkFalse);
436   RunPrimeVectorTest('RemoveElementAt_RemovesElement',
      ↪  RemoveElementAt_RemovesElement);
437   RunPrimeVectorTest('RemoveElementAtOutOfBounds_OkFalse',
      ↪  RemoveElementAtOutOfBounds_OkFalse);
438   RunPrimeVectorTest('NonPrime_DoesNotAddAndInsert',
      ↪  NonPrime_DoesNotAddAndInsert);
439   RunPrimeVectorTest('NegativeNumber_AddsAndInserts',
      ↪  NegativeNumber_AddsAndInserts);
440   WriteLn('All tests passed');
441 END. (* TestVector *)
```

1.3.2 TestVector.pas

```
1  PROGRAM TestStack;
2
3  USES
4  UCardinalStack;
5
6  TYPE
7    test = FUNCTION : BOOLEAN;
8
9  FUNCTION InitialStack_IsEmpty: BOOLEAN;
10 VAR
11   s: CardinalStack;
12 BEGIN (* InitialStack_IsEmpty *)
13   s.Init();
14   InitialStack_IsEmpty := s.IsEmpty();
15   s.Done();
16 END; (* InitialStack_IsEmpty *)
17
```

```
18  FUNCTION PushOne_IsNotEmpty: BOOLEAN;
19  VAR
20    s: CardinalStack;
21    pushOk: BOOLEAN;
22  BEGIN (* PushOne_IsNotEmpty *)
23    s.Init();
24    s.Push(1, pushOk);
25    PushOne_IsNotEmpty := NOT s.IsEmpty()
26                          AND pushOk;
27    s.Done();
28  END; (* PushOne_IsNotEmpty *)
29
30  FUNCTION PushOnePopOne_IsEmpty: BOOLEAN;
31  VAR
32    s: CardinalStack;
33    pushOk, popOk: BOOLEAN;
34    value: INTEGER;
35  BEGIN (* PushOnePopOne_IsEmpty *)
36    s.Init();
37    s.Push(1, pushOk);
38    s.Pop(value, popOk);
39    PushOnePopOne_IsEmpty := s.IsEmpty()
40                             AND pushOk
41                             AND popOk
42                             AND (value = 1);
43    s.Done();
44  END; (* PushOnePopOne_IsEmpty *)
45
46  FUNCTION PushNegative_DoesNotPush: BOOLEAN;
47  VAR
48    s: CardinalStack;
49    pushOk: BOOLEAN;
50  BEGIN (* PushNegative_DoesNotPush *)
51    s.Init();
52    s.Push(-1, pushOk);
53    PushNegative_DoesNotPush := s.IsEmpty()
54                                AND NOT pushOk;
55    s.Done();
56  END; (* PushNegative_DoesNotPush *)
57
58  PROCEDURE RunTest(NAME: STRING; t: test);
59  BEGIN (* RunTest *)
60    IF (t()) THEN
61      BEGIN (* IF *)
62        WriteLn('PASSED - ', name);
63      END (* IF *)
64    ELSE
```

```
65      BEGIN (* ELSE *)
66        WriteLn('FAILED - ', name);
67        Halt(1);
68      END; (* ELSE *)
69  END; (* RunTest *)
70
71  BEGIN (* TestStack *)
72    WriteLn('CardinalStack:');
73    RunTest('InitialStack_IsEmpty', InitialStack_IsEmpty);
74    RunTest('PushOne_IsNotEmpty', PushOne_IsNotEmpty);
75    RunTest('PushOnePopOne_IsEmpty', PushOnePopOne_IsEmpty);
76    RunTest('PushNegative_DoesNotPush', PushNegative_DoesNotPush);
77    WriteLn('All tests passed');
78  END. (* TestStack *)
```

1.3.3 TestVector.pas

```
1   PROGRAM TestQueue;
2
3   USES
4   UEvenQueue;
5
6   TYPE
7     test = FUNCTION : BOOLEAN;
8
9   FUNCTION InitialQueue_IsEmpty: BOOLEAN;
10  VAR
11    q: EvenQueue;
12  BEGIN (* InitialQueue_IsEmpty *)
13    q.Init();
14    InitialQueue_IsEmpty := q.IsEmpty();
15    q.Done();
16  END; (* InitialQueue_IsEmpty *)
17
18  FUNCTION EnqueueOne_IsNotEmpty: BOOLEAN;
19  VAR
20    q: EvenQueue;
21    enqueueOk: BOOLEAN;
22  BEGIN (* EnqueueOne_IsNotEmpty *)
23    q.Init();
24    q.Enqueue(2, enqueueOk);
25    EnqueueOne_IsNotEmpty := NOT q.IsEmpty()
26                             AND enqueueOk;
27    q.Done();
28  END; (* EnqueueOne_IsNotEmpty *)
29
30  FUNCTION EnqueueOneDequeueOne_IsEmpty: BOOLEAN;
31  VAR
```

```
32    q: EvenQueue;
33    enqueueOk: BOOLEAN;
34    dequeueOk: BOOLEAN;
35    value: INTEGER;
36  BEGIN (* EnqueueOneDequeueOne_IsEmpty *)
37    q.Init();
38    q.Enqueue(2, enqueueOk);
39    q.Dequeue(value, dequeueOk);
40    EnqueueOneDequeueOne_IsEmpty := q.IsEmpty()
41                                    AND enqueueOk
42                                    AND dequeueOk
43                                    AND (value = 2);
44    q.Done();
45  END; (* EnqueueOneDequeueOne_IsEmpty *)
46
47  FUNCTION EnqueueUneven_DoesNotEnqueue: BOOLEAN;
48  VAR
49    q: EvenQueue;
50    enqueueOk: BOOLEAN;
51  BEGIN (* EnqueueUneven_DoesNotEnqueue *)
52    q.Init();
53    q.Enqueue(3, enqueueOk);
54    EnqueueUneven_DoesNotEnqueue := NOT enqueueOk
55                                    AND q.IsEmpty();
56    q.Done();
57  END; (* EnqueueUneven_DoesNotEnqueue *)
58
59  PROCEDURE RunTest(NAME: STRING; t: test);
60  BEGIN (* RunTest *)
61    IF (t()) 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 (* TestQueue *)
73    WriteLn('EvenQueue:');
74    RunTest('InitialQueue_IsEmpty', InitialQueue_IsEmpty);
75    RunTest('EnqueueOne_IsNotEmpty', EnqueueOne_IsNotEmpty);
76    RunTest('EnqueueOneDequeueOne_IsEmpty', EnqueueOneDequeueOne_IsEmpty);
77    RunTest('EnqueueUneven_DoesNotEnqueue', EnqueueUneven_DoesNotEnqueue);
78    WriteLn('All tests passed');
```

```
79 END. (* TestQueue *)
```

1.3.4 Testskript

```
1 #!/bin/bash
2 ../bin/TestVector;
3 echo;
4 ../bin/TestStack;
5 echo;
6 ../bin/TestQueue;
```

1.4 Testergebnisse

1.4.1 Ausgabe des Testskripts

Listing 1: TestResult.txt

```
Vector:
PASSED - InitialVector_IsEmpty
PASSED - ClearVector_IsEmpty
PASSED - AddElement_IncreasesSize
PASSED - AddElementsOverCapacity_OkFalse
PASSED - GetElementAt_ReturnsCorrectElement
PASSED - GetElementAtOutOfBounds_OkFalse
PASSED - InsertElementAt_AddsElementToPosition
PASSED - InsertElementAtGreaterThanCapacity_DoesNotInsert
PASSED - InsertElementAtLessThanOne_AddsToFront
PASSED - InsertElementAtGreaterThanSize_AddsToEnd
PASSED - InsertElementAtAlreadyFull_OkFalse
PASSED - RemoveElementAt_RemovesElement
PASSED - RemoveElementAtOutOfBounds_OkFalse
PASSED - NonPrime_AddsAndInserts
PASSED - NegativeNumber_AddsAndInserts

NaturalVector:
PASSED - InitialVector_IsEmpty
PASSED - ClearVector_IsEmpty
PASSED - AddElement_IncreasesSize
PASSED - AddElementsOverCapacity_OkFalse
PASSED - GetElementAt_ReturnsCorrectElement
PASSED - GetElementAtOutOfBounds_OkFalse
PASSED - InsertElementAt_AddsElementToPosition
PASSED - InsertElementAtGreaterThanCapacity_DoesNotInsert
PASSED - InsertElementAtLessThanOne_AddsToFront
PASSED - InsertElementAtGreaterThanSize_AddsToEnd
PASSED - InsertElementAtAlreadyFull_OkFalse
PASSED - RemoveElementAt_RemovesElement
PASSED - RemoveElementAtOutOfBounds_OkFalse
PASSED - NonPrime_AddsAndInserts
PASSED - NegativeNumber_DoesNotAddAndInsert

PrimeVector:
PASSED - InitialVector_IsEmpty
PASSED - ClearVector_IsEmpty
PASSED - AddElement_IncreasesSize
PASSED - AddElementsOverCapacity_OkFalse
PASSED - GetElementAt_ReturnsCorrectElement
PASSED - GetElementAtOutOfBounds_OkFalse
PASSED - InsertElementAt_AddsElementToPosition
```

```
PASSED - InsertElementAtGreaterThanCapacity_DoesNotInsert
PASSED - InsertElementAtLessThanOne_AddsToFront
PASSED - InsertElementAtGreaterThanSize_AddsToEnd
PASSED - InsertElementAtAlreadyFull_OkFalse
PASSED - RemoveElementAt_RemovesElement
PASSED - RemoveElementAtOutOfBounds_OkFalse
PASSED - NonPrime_DoesNotAddAndInsert
PASSED - NegativeNumber_AddsAndInserts
All tests passed

CardinalStack:
PASSED - InitialStack_IsEmpty
PASSED - PushOne_IsNotEmpty
PASSED - PushOnePopOne_IsEmpty
PASSED - PushNegative_DoesNotPush
All tests passed

EvenQueue:
PASSED - InitialQueue_IsEmpty
PASSED - EnqueueOne_IsNotEmpty
PASSED - EnqueueOneDequeueOne_IsEmpty
PASSED - EnqueueUneven_DoesNotEnqueue
All tests passed
```