☒ **Gr. 1,** Dr. G. Kronberger ☐ **Gr. 2,** Dr. H. Gruber ☐ **Gr. 3,** Dr. D. Auer

Name PAPESH Konstantin **Aufwand in h** 10

Punkte 76 **Kurzzeichen Tutor / Übungsleiter** L. S. / _____

1. „Behälter“ *Vector* als ADS und ADT (10 + 6 Punkte)

Aus dem ersten Semester wissen Sie ja (hoffentlich noch), 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 *V_ADS.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 [`**`Get`**`|`**`Set`**`]`**`ElementAt`**`(pos: INTEGER; [VAR] val: INTEGER)`
liefert/setzt an der Stelle *pos* den Wert *val*.

`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 Fehlersituation 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 *V_ADT.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*) (5 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 *Q_ADS.pas*).

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

3. Und zum Abschluss noch schnell ein neuer Kellerspeicher (*Stack*) (3 Punkte)

Abschließend ist noch ein weiteres Modul für einen Kellerspeicher in Form eines abstrakten Daten**typs** (in *S_ADT.pas*) zu implementieren, welcher sich des Behälters aus 1.b) bedient.

Mindestens die folgenden Operationen müssen zur Verfügung gestellt werden: `IsEmpty`, `Push` und `Pop`.

ADF2x & PRO2X Algorithmen & Datenstrukturen und OOP – SS 2018
Übungsabgabe 5

Konstantin Papesh

1. Mai 2018

Zusammenfassung

In dieser Übung wird eine Vorstufe von Klassen vorgestellt. Dabei werden Datentypen verwendet, welche mithilfe von Informationhiding und Datenkapslung für eine Instanzierung der einzelnen Exemplare sorgt.

5.1 Behälter Vector als ADS und ADT

5.1.1 Lösungsidee

Um das Problem eines undynamischen Behälters zu lösen, setzen wir einen flexiblen Behälter ein, der die Speicherverwaltung im Hintergrund übernimmt. Der Nutzer fügt lediglich zu dem Behälter hinzu, ändert/ließt Werte oder entfernt sie, um die Vergrößerung des Vectors muss er sich nicht kümmern.

a)

Dafür implementieren wir einen Behälter *Vector*, welcher Integer aufnimmt, mithilfe des Befehls *add*. Intern wird bei *add* gleich kontrolliert, wie groß die Kapazität des momentanen Vectors ist. Sollte die Kapazität des momentanen Vectors ausgereizt sein, wird dieser automatisch vergrößert. Dabei wird ein neuer Vector erstellt, die Werte des alten Vectors an die selbe Position im neuen Vector geschrieben und letztendlich der alte Vector gelöscht und der Pointer für den Vector auf den neuen Vector gesetzt. So ist ein immer befüllbarer Vector garantiert.

+ Zugriffsprüfungen bei Get-/Set-/Remove-Funktionen

b)

Um *a)* als Daten*typ* ausführen zu können müssen einige Änderungen am Code vorgenommen werden. Dabei wird statt direkt mit der Datenstruktur über Pointer gearbeitet. Im Hauptprogramm werden die bereits gecodeten Funktionen aufgerufen, jedoch mit einem zusätzlichen Parameter *s : stackPtr*, welcher den Pointer zu dem jeweiligen Vector

+ Name -0,5

liefert. Dadurch können mehrere Vectoren gleichzeitig parallel exisitieren und bearbeitet werden. Intern (im Modul, in diesem Fall *V_ADT.pas*) ist an dieser Speicheradresse ein Typ mit dem Pointer zum eigentlichen Array, die momentane Kapazität des Arrays und die momentane Größe dieses. Da dieser Typ ausserhalb jedoch nicht bekannt ist, können die Variablen in diesem nicht von ausserhalb bearbeitet werden.

5.1.2 Implementierung

Listing 5.1: V_ADS.pas

```
1  unit V_ADS;
2
3  interface
4  procedure errorIfOutOfRange(pos : integer);
5  procedure add(val : integer);
6  procedure getElementAt(pos : integer; var val : integer);
7  procedure setElementAt(pos : integer; val : integer);
8  procedure removeElementAt(pos : integer);
9  function size : integer;
10 function capacity : integer;
11 function isEmpty : boolean;
12 procedure init;
13 procedure disposeStack;
14 procedure reallocStack;
15
16 implementation
17 type
18     intArray = array [1..1] of integer;
19
20 VAR
21     arrPtr : ^intArray;
22     capacityCount : integer;
23     top : integer; (* index of top element *)
24
25 procedure init;
26 begin
27     if(arrPtr <> NIL) then begin
28         writeln('Can''t initialize non-empty stack!');
29         halt;
30     end;
31     top := 0;
32     capacityCount := 2;
33     GetMem(arrPtr, SIZEOF(integer) * capacityCount);
34 end;
35
36 procedure errorIfOutOfRange(pos : integer);
37 begin
38     if pos > top then begin
39         writeln('Pos out of range!');
40         halt;
41     end;
42 end;
43 procedure add(val : integer);
44 begin
45     if top >= capacityCount then
```

+ Syntax-Highlighting -0,5

zum Typ hinzufügen -0,5

Magic-Number (unflexibel) -0,5

etwas harte Fehlerbehandlung

```
46          reallocStack;
47      inc(top);
48      (*$R-*)
49      arrPtr^[top] := val;
50      (*$R+*)
51  end;
52
53  procedure getElementAt(pos : integer; var val : integer);
54  begin
55      errorIfOutOfRange(pos);
56      (*$R-*)
57      val := arrPtr^[pos];
58      (*$R+*)
59  end;
60
61  procedure setElementAt(pos : integer; val : integer);
62  begin
63      errorIfOutOfRange(pos);
64      (*$R-*)
65      arrPtr^[pos] := val;
66      (*$R+*)
67  end;
68
69  procedure removeElementAt(pos : integer);
70  var
71      element : integer;
72  begin
73      errorIfOutOfRange(pos);
74      element := pos + 1;
75      while element <= top do begin
76          (*$R-*)
77          arrPtr^[element - 1] := arrPtr^[element];
78          (*$R+*)
79          inc(element);
80      end;
81      (*$R-*)
82      arrPtr^[top] := 0;
83      (*$R+*)
84      dec(top);
85  end;
86
87  function size : integer;
88  begin
89      size := top;
90  end;
91
92  function capacity : integer;
93  begin
94      capacity := capacityCount;
95  end;
96
97  function isEmpty : boolean;
98  begin
99      isEmpty := top = 0;
100 end;
101
102 procedure disposeStack;
```

```
103 begin
104     if arrPtr = NIL then begin
105         writeln('Can''t dispose a uninitialized stack!');
106         halt;
107     end;
108     freeMem(arrPtr, SIZEOF(integer) * capacityCount);
109     arrPtr := NIL;
110 end;
111
112 procedure reallocStack;
113 var
114     newArray : ^intArray;
115     i : integer;
116 begin
117     getMem(newArray, SIZEOF(INTEGER) * 2 * capacityCount);
118     for i := 1 to top do begin
119         (*$R-*)
120         newArray^[i] := arrPtr^[i];
121         (*$R+*)
122     end;
123     freeMem(arrPtr, SIZEOF(integer) * capacityCount);
124     capacityCount := 2 * capacityCount;
125     arrPtr := newArray;
126 end;
127
128 begin
129     arrPtr := NIL;
130 end.
```

Magic-Number (unflexibel)
- 0,5

Listing 5.2: TestModVector.pas

```
 1 program TestModVector;
 2 uses V_ADS;
 3
 4 var i : integer;
 5     tVal : integer;
 6 begin
 7     init;
 8     for i := 1 to 50 do begin
 9         add(i);
10     end;
11
12     writeln('IsEmpty?', IsEmpty);
13     writeln('Size', size);
14     writeln('Capacity', capacity);
15     getElementAt(2, tVal);
16     writeln('Second element', tVal);
17     setElementAt(2, tVal + 5);
18     getElementAt(2, tVal);
19     writeln('Second element', tVal);
20     RemoveElementAt(2);
21     getElementAt(2, tVal);
22     writeln('Second element', tVal);
23     writeln('Size', size);
24     writeln('Capacity', capacity);
25     RemoveElementAt(size);
26 end.
```

Listing 5.3: V_ADT.pas

```
1  unit V_ADT;
2
3  interface
4  type
5      intArray = array [1..1] of integer;
6      stackPtr = POINTER;
7
8  procedure add(s : stackPtr; val : integer);
9  procedure getElementAt(s : stackPtr; pos : integer; var val : integer);
10 procedure setElementAt(s : stackPtr; pos : integer; val : integer);
11 procedure removeElementAt(s : stackPtr; pos : integer);
12 function size(s : stackPtr) : integer;
13 function capacity(s : stackPtr) : integer;
14 function isEmpty(s : stackPtr) : boolean;
15 procedure init(var s : stackPtr);
16 procedure disposeStack(var s : stackPtr);
17
18 implementation
19 type
20     internalStackPtr = ^stack;
21     stack = record
22                 arrPtr : ^intArray;
23                 capacityCount : integer;
24                 top : integer; (* index of top element *)
25             end;
26 procedure reallocStack(var s : stackPtr); FORWARD;
27 procedure errorIfOutOfRange(s : stackPtr; pos : integer); FORWARD;
28
29 procedure init(var s : stackPtr);
30 var isPtr : internalStackPtr;
31 begin
32     isPtr := internalStackPtr(s);
33     if(s <> NIL) then begin
34         writeln('Can''t initialize non-empty stack!');
35         halt;
36     end;
37     new(isPtr);
38     isPtr^.top := 0;
39     isPtr^.capacityCount := 16;
40     GetMem(isPtr^.arrPtr, SIZEOF(integer) * isPtr^.capacityCount);
41     s := isPtr;
42 end;
43
44 procedure errorIfOutOfRange(s : stackPtr; pos : integer);
45 var isPtr : internalStackPtr;
46 begin
47     isPtr := internalStackPtr(s);
48     if pos > isPtr^.top then begin
49         writeln('Pos out of range!');
50         halt;
51     end;
52 end;
53
54 procedure add(s : stackPtr; val : integer);
55 var isPtr : internalStackPtr;
```

einheitlich - 0,5

schlechten Variablennamen - 0,5

wir sind noch beim Vektor - 0,5

viel besser

Magic-Number - 0,5

```
56 begin
57     isPtr := internalStackPtr(s);
58     if isPtr^.top >= isPtr^.capacityCount then begin
59         reallocStack(s);
60     end;
61     inc(isPtr^.top);
62     (*$R-*)
63     isPtr^.arrPtr^[isPtr^.top] := val;
64     (*$R+*)
65 end;
66
67 procedure getElementAt(s : stackPtr; pos : integer; var val : integer);
68 var isPtr : internalStackPtr;
69 begin
70     isPtr := internalStackPtr(s);
71     errorIfOutOfRange(isPtr, pos);
72     (*$R-*)
73     val := isPtr^.arrPtr^[pos];
74     (*$R+*)
75 end;
76
77 procedure setElementAt(s : stackPtr; pos : integer; val : integer);
78 var isPtr : internalStackPtr;
79 begin
80     isPtr := internalStackPtr(s);
81     errorIfOutOfRange(isPtr, pos);
82     (*$R-*)
83     isPtr^.arrPtr^[pos] := val;
84     (*$R+*)
85 end;
86
87 procedure removeElementAt(s : stackPtr; pos : integer);
88 var
89     isPtr : internalStackPtr;
90     element : integer;
91 begin
92     isPtr := internalStackPtr(s);
93     errorIfOutOfRange(isPtr, pos);
94     element := pos + 1;
95     while element <= isPtr^.top do begin
96         (*$R-*)
97         isPtr^.arrPtr^[element - 1] := isPtr^.arrPtr^[element];
98         (*$R+*)
99         inc(element);
100    end;
101    (*$R-*)
102    isPtr^.arrPtr^[isPtr^.top] := 0;
103    (*$R+*)
104    dec(isPtr^.top);
105 end;
106
107 function size(s : stackPtr) : integer;
108 var isPtr : internalStackPtr;
109 begin
110    isPtr := internalStackPtr(s);
111    size := isPtr^.top;
112 end;
```

```
113
114 function capacity(s : stackPtr) : integer;
115 var isPtr : internalStackPtr;
116 begin
117     isPtr := internalStackPtr(s);
118     capacity := isPtr^.capacityCount;
119 end;
120
121 function isEmpty(s : stackPtr) : boolean;
122 var isPtr : internalStackPtr;
123 begin
124     isPtr := internalStackPtr(s);
125     isEmpty := isPtr^.top = 0;
126 end;
127
128 procedure disposeStack(var s : stackPtr);
129 var isPtr : internalStackPtr;
130 begin
131     if s = NIL then begin
132         writeln('Can''t dispose a uninitialized stack!');
133         halt;
134     end;
135     isPtr := internalStackPtr(s);
136     freeMem(isPtr^.arrPtr, SIZEOF(integer) * isPtr^.capacityCount);
137     isPtr^.arrPtr := NIL;
138     dispose(isPtr);
139     s := NIL;
140 end;
141
142 procedure reallocStack(var s : stackPtr);
143 var isPtr : internalStackPtr;
144     newArray : ^intArray;
145     i : integer;
146 begin
147     isPtr := internalStackPtr(s);
148     getMem(newArray, SIZEOF(INTEGER) * 2 * isPtr^.capacityCount);
149     for i := 1 to isPtr^.top do begin
150         (*$R-*)
151         newArray^[i] := isPtr^.arrPtr^[i];
152         (*$R+*)
153     end;
154     freeMem(isPtr^.arrPtr, SIZEOF(integer) * isPtr^.capacityCount);
155     isPtr^.capacityCount := 2 * isPtr^.capacityCount;
156     isPtr^.arrPtr := newArray;
157 end;
158
159 begin
160 end.
```

Listing 5.4: TestModVectorADT.pas

```
1 program TestModVectorADT;
2 uses V_ADT;
3
4 var i : integer;
5     tVal : integer;
6     s0 : stackPtr;
```

```
 7 begin
 8     init(s0);
 9     for i := 1 to 50 do begin
10         add(s0, i);
11     end;
12
13     writeln('IsEmpty?', IsEmpty(s0));
14     writeln('Size', size(s0));
15     writeln('Capacity', capacity(s0));
16     getElementAt(s0, 2, tVal);
17     writeln('Second element', tVal);
18     setElementAt(s0, 2, tVal + 5);
19     getElementAt(s0, 2, tVal);
20     writeln('Second element', tVal);
21     RemoveElementAt(s0, 2);
22     getElementAt(s0, 2, tVal);
23     writeln('Second element', tVal);
24     writeln('Size', size(s0));
25     writeln('Capacity', capacity(s0));
26     RemoveElementAt(s0, size(s0));
27 end.
```

5.1.3 Ausgabe

Die Ausgaben von *a)* und *b)* sind ident, da die Änderungen am Code nur intern sind und sich in der Ausgabe nicht wiederspiegeln.

```
IsEmpty = FALSE
Size = 50
Capacity = 64
Second element = 2
Add 5 to second element
Second element (should now be 7) = 7
Now remove second element --> third element shifts into its place.
Thus second element should now be '3' and size should be one less.
Second element = 3
Size = 49
Capacity = 64
```

Abbildung 5.1: Befüllung und Bearbeitung des Arrays

5.1.4 Auswertung

5.2 Warteschlange (Queue)

5.2.1 Lösungsidee

Grundsätzlich entspricht die Warteschlange fast der ausgearbeiteten Datenstruktur von 5.1.1, lediglich der Zugriff für das Abspeichern und Herausholen wird abgewandelt. Statt Elemente einzeln abrufen, bearbeiten oder löschen zu können, funktioniert die Warteschlange ähnlich einem "Kanal"nach dem *FIFO-Prinzip*[1]. Hinzugefügte Elemente *Enqueue* werden hinten hineingedrückt, mit *Dequeue* wird vorne das erste Element entnommen.
Folglich wird gleich wie in 5.1.1 zuerst eine Queue initialisiert. Danach kann diese nach belieben mit Integerzahlen befüllt werden mit *Enqueue*. Dabei wird das aktuell hinzugefügte Element einfach an das Ende des Arrays geschrieben. Wird *Dequeue* aufgerufen,

[1] First In - First Out

wird das erste Element des Arrays zurückgeliefert und gleichzeitig auch aus dem Array entfernt. Dann wird das ganze Array nach Links geshiftet, damit der erste Eintrag des Arrays wieder einen Wert enthält. Es kann nur so oft ein Wert entnommen werden als auch Werte hineingeschrieben wurden. Wird versucht, mehr zu entnehmen wird eine Fehlermeldung ausgegeben und das Programm beendet sich.

5.2.2 Implementierung

Listing 5.5: Q_ADS.pas

```
 1 unit Q_ADS;
 2
 3 interface
 4 procedure enqueue(e : integer);
 5 function dequeue : integer;
 6 function isEmpty : boolean;
 7 procedure init;
 8 procedure disposeQueue;
 9 procedure reallocQueue;
10 procedure removeElementAt(pos : integer);
11
12 implementation
13 type
14     intArray = array [1..1] of integer;
15
16 VAR
17     arrPtr : ^intArray;
18     capacity : integer;
19     top : integer; (* index of top element *)
20
21 procedure init;
22 begin
23     if(arrPtr <> NIL) then begin
24         writeln('Can''t initialize non-empty queue!');
25         halt;
26     end;
27     top := 0;
28     capacity := 10;
29     GetMem(arrPtr, SIZEOF(integer) * capacity);
30 end;
31
32 procedure enqueue(e : integer);
33 begin
34     if top >= capacity then
35         reallocQueue;
36     inc(top);
37     (*$R-*)
38     arrPtr^[top] := e;
39     (*$R+*)
40 end;
41
42 function dequeue : integer;
43 begin
44     if isEmpty then begin
45         writeln('Queue is empty');
```

```
46          halt;
47      end;
48      (*$R-*)
49      dequeue := arrPtr^[1];
50      (*$R+*)
51      removeElementAt(1);
52  end;
53
54  function isEmpty : boolean;
55  begin
56      isEmpty := top = 0;
57  end;
58
59  procedure disposeQueue;
60  begin
61      if arrPtr = NIL then begin
62          writeln('Can''t dispose a uninitialized queue!');
63          halt;
64      end;
65      FreeMem(arrPtr, SIZEOF(integer) * capacity);
66      arrPtr := NIL;
67  end;
68
69  procedure reallocQueue;
70  var
71      newArray : ^intArray;
72      i : integer;
73  begin
74      GetMem(newArray, SIZEOF(INTEGER) * 2 * capacity);
75      for i := 1 to top do begin
76          (*$R-*)
77          newArray^[i] := arrPtr^[i];
78          (*$R+*)
79      end;
80      FreeMem(arrPtr, SIZEOF(integer) * capacity);
81      capacity := 2 * capacity;
82      arrPtr := newArray;
83  end;
84
85  procedure removeElementAt(pos : integer);
86  var
87      element : integer;
88  begin
89      element := pos + 1;
90      while element <= top do begin
91          (*$R-*)
92          arrPtr^[element - 1] := arrPtr^[element];
93          (*$R+*)
94          inc(element);
95      end;
96      (*$R-*)
97      arrPtr^[top] := 0;
98      (*$R+*)
99      dec(top);
100 end;
101
102 begin
```

```
103     arrPtr := NIL;
104 end.
```

Listing 5.6: TestModQueue.pas

```
 1 program TestModQueue;
 2 uses Q_ADS;
 3
 4 var i : integer;
 5 begin
 6     init;
 7     for i := 1 to 50 do begin
 8         enqueue(i);
 9     end;
10
11     writeln('IsEmpty?', IsEmpty);
12     writeln('Elements:');
13     while not IsEmpty do begin
14         writeln(dequeue);
15     end;
16     writeln('IsEmpty? ', IsEmpty);
17     disposeQueue;
18 end.
```

5.2.3 Ausgabe

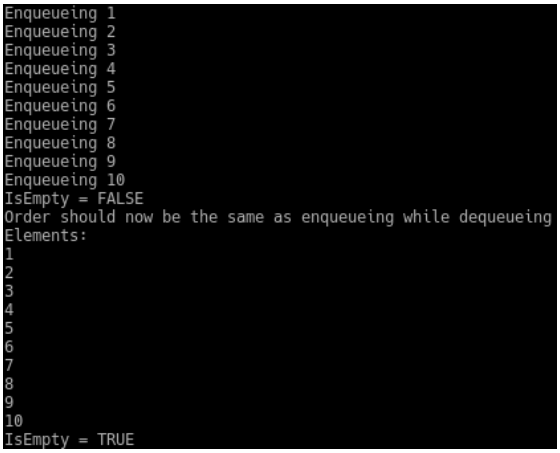

Abbildung 5.2: Enqueueing und Dequeueing

5.3 Kellerspeicher (Stack)

5.3.1 Lösungsidee

Der Kellerspeicher als Daten*typ* entspricht einer Mischung aus 5.1.1 und 5.2. Die Implementierung des Datentyps entspricht 5.1.1. Die Funktionalität von *Push* ist Ident mit *Enqueue* von 5.2. Der Unterschied von *Pop* zu *Dequeue* ist, dass *Pop* den letzten eingefügten Wert zurückliefert anstatt den ersten wie bei *Dequeue*. Also das LIFO-Prinzip[2].

[2] Last In - First Out

Durch diese Funktionalität von *Pop* muss auch das Array nicht mehr geshiftet werden, lediglich die Größe muss um *1* verkleinert werden.

5.3.2 Implementierung

Listing 5.7: S_ADT.pas

```
1  unit S_ADT;
2
3  interface
4  type
5      intArray = array [1..1] of integer;
6      stackPtr = POINTER;
7
8  procedure push(s: stackPtr; e: integer);
9  procedure pop(s : stackPtr; var e : integer);
10 function isEmpty(s: stackPtr) : boolean;
11 procedure init(var s : stackPtr);
12 procedure disposeStack(var s : stackPtr);
13
14 implementation
15 type
16     internalStackPtr = ^stack;
17     stack = record
18                 arrPtr : ^intArray;
19                 capacity : integer;
20                 top : integer; (* index of top element *)
21             end;
22 procedure reallocStack(s : stackPtr); FORWARD;
23
24 procedure init(var s : stackPtr);
25 var isPtr : internalStackPtr;
26 begin
27     isPtr := internalStackPtr(s);
28
29     if s <> NIL then begin
30         writeln('Call disposeStack first you maniac!');
31         halt;
32     end;
33     new(isPtr);
34     isPtr^.top := 0;
35     isPtr^.capacity := 10;
36     GetMem(isPtr^.arrPtr, SIZEOF(integer) * isPtr^.capacity);
37     s := isPtr;
38 end;
39
40 procedure push(s : stackPtr; e : integer);
41 var isPtr : internalStackPtr;
42 begin
43     isPtr := internalStackPtr(s);
44     if isPtr^.top >= isPtr^.capacity then begin
45         reallocStack(s);
46     end;
47     inc(isPtr^.top);
48     (*$R-*)
49     isPtr^.arrPtr^[isPtr^.top] := e;
```

```
50      (*$R+*)
51  end;
52
53  procedure pop(s : stackPtr; var e : integer);
54  var isPtr : internalStackPtr;
55  begin
56      isPtr := internalStackPtr(s);
57      if isEmpty(s) then begin
58          writeln('Stack is empty');
59          halt;
60      end;
61      dec(isPtr^.top);
62      (*$R-*)
63      e := isPtr^.arrPtr^[isPtr^.top+1];
64      (*$R+*)
65  end;
66
67  function isEmpty(s : stackPtr) : boolean;
68  begin
69      isEmpty := internalStackPtr(s)^.top = 0;
70  end;
71
72  procedure disposeStack(var s : stackPtr);
73  var isPtr : internalStackPtr;
74  begin
75      if s = NIL then begin
76          writeln('Stack is not initialized you moron!');
77          halt;
78      end;
79      isPtr := internalStackPtr(s);
80      FreeMem(isPtr^.arrPtr, SIZEOF(integer) * isPtr^.capacity);
81      isPtr^.arrPtr := NIL;
82      dispose(isPtr);
83      s := NIL;
84  end;
85
86  procedure reallocStack(s : stackPtr);
87  var isPtr : internalStackPtr;
88      newArray : ^intArray;
89      i : integer;
90  begin
91      isPtr := internalStackPtr(s);
92      GetMem(newArray, SIZEOF(INTEGER) * 2 * isPtr^.capacity);
93      for i := 1 to isPtr^.top do begin
94          (*$R-*)
95          newArray^[i] := isPtr^.arrPtr^[i];
96          (*$R+*)
97      end;
98      FreeMem(isPtr^.arrPtr, SIZEOF(integer) * isPtr^.capacity);
99      isPtr^.capacity := 2 * isPtr^.capacity;
100     isPtr^.arrPtr := newArray;
101 end;
102
103 begin
104 end.
```

Listing 5.8: TestModStack.pas

```
1 program TestModStack;
2 uses S_ADT;
3
4 var i : integer;
5     s0, s1 : stackPtr;
6 begin
7     init(s0);
8     init(s1);
9     for i := 1 to 50 do begin
10        if(Odd(i)) then
11            push(s0, i)
12        else
13            push(s1, i);
14    end;
15
16    writeln('IsEmpty(s0) = ', IsEmpty(s0));
17    writeln('IsEmpty(s1) = ', IsEmpty(s1));
18    writeln('Popping s0 (all odd numbers) reversed (49 - 1)');
19    writeln('Elements:');
20    while not IsEmpty(s0) do begin
21        pop(s0, i);
22        writeln(i);
23    end;
24    disposeStack(s0);
25    disposeStack(s1);
26 end.
```

5.3.3 Ausgabe

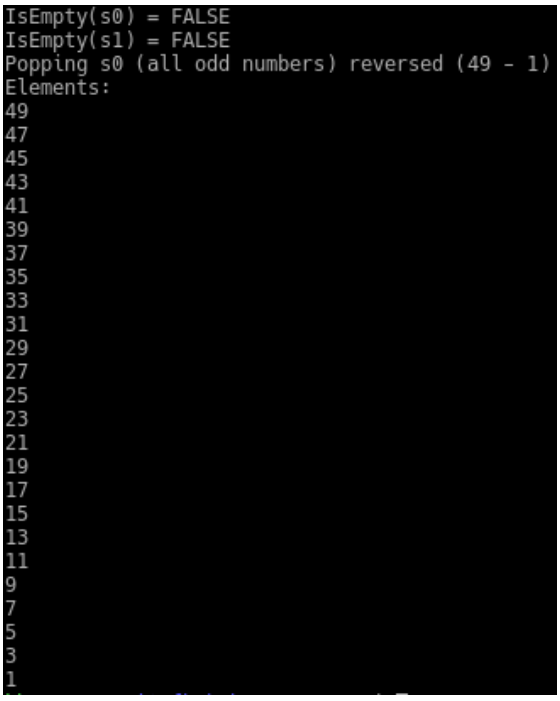

Abbildung 5.3: Beispiel für Pushing und Popping mit zwei Stacks