**Abgabetermin: Mi, 21. 04. 2021**

☐ **Gr. 1,** Dr. S. Wagner **Name** Angelos Angelis **Aufwand in h** 9
☒ **Gr. 2,** Dr. D. Auer
☐ **Gr. 3,** Dr. G. Kronberger **Punkte** _____ **Kurzzeichen Tutor / Übungsleiter*in** _____ / _____

## 1. „Behälter“ *Vector* als ADS und ADT (12 + 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 *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.

1) **Lösungsidee:**
   a. `PROCEDURE Add(val: INTEGER);`
      i. Man muss ein Element hinten hinzufügen, wobei falls die größe des Arrays überschritten wird das Array mit Hilfe der Prozedur DoubleSize verdoppelt wird
   b. `PROCEDURE SetElementAt(pos: INTEGER; val: INTEGER);`
      i. Man setzt an eine ausgewählten position einen Wert ein. An sich selbstverständlich man sollte jedoch vorher überprüfen ob die angegebene Position gültig ist.
   c. `FUNCTION ElementAt(pos: INTEGER): INTEGER;`
      i. Liefert den Wert an der gewünschten Position zurück. Hier sollt man auch überprüfen ob die Position gültig ist
   d. `PROCEDURE RemoveElementAt(pos: INTEGER);`
      i. Löscht das Element an der gewünschten Stelle. Hier sollte man ebenfalls überprüfen ob die Position gültig ist. Nach dem Löschen sollen ebenfalls alle folgenden Elemente um eine Stelle nach hinten verschoben werden
   e. `FUNCTION Size: INTEGER;`
      i. Liefert die Anzahl der besetzten Zahlen im Array.
   f. `FUNCTION Capacity: INTEGER;`
      i. Liefert die insgesammte Größe des Feldes

2) **Testfälle**

Testfall1)

```
Loaded VectorADS
Add 20 ints to Vector
Writing Vector its Size and Capacity
1: 1
2: 2
3: 3
4: 4
5: 5
6: 6
7: 7
8: 8
9: 9
10: 10
11: 11
12: 12
13: 13
14: 14
15: 15
16: 16
17: 17
18: 18
19: 19
20: 20
Size: 20
Capacity: 20
```

Testfall2)

```
Loaded VectorADS
Add 5 ints to Vector when Capacity = 10
Writing Vector its Size and Capacity
1: 1
2: 2
3: 3
4: 4
5: 5
6: 0
7: 0
8: 0
9: 0
10: 0
Size: 5
Capacity: 10
```

Testfall3)

```
Loaded VectorADS
Add 20 ints to Vector
SetElement at pos 11 to val 88
RemoveElement At 12
Writing Vector its Size and Capacity
1: 1
2: 2
3: 3
4: 4
5: 5
6: 6
7: 7
8: 8
9: 9
10: 10
11: 88
12: 13
13: 14
14: 15
15: 16
16: 17
17: 18
18: 19
19: 20
20: 0
Size: 19
Capacity: 20
```

## 3) Quellcode

### UNIT:

```
UNIT VectorADS;

INTERFACE

  PROCEDURE Assert(cond: BOOLEAN; msg: STRING);
  PROCEDURE InitVector;
  PROCEDURE DoubleSize;
  PROCEDURE Add(val: INTEGER);
  PROCEDURE SetElementAt(pos: INTEGER; val: INTEGER);
  FUNCTION ElementAt(pos: INTEGER): INTEGER;
  PROCEDURE RemoveElementAt(pos: INTEGER);
  FUNCTION Size: INTEGER;
  FUNCTION Capacity: INTEGER;
  PROCEDURE WriteVector;


IMPLEMENTATION
  CONST
    startSize = 10;
  TYPE
    Vector = ARRAY [1..1] OF INTEGER;
  VAR
    Vsize: INTEGER;
    VPtr: ^Vector;

  PROCEDURE Assert(cond: BOOLEAN;msg: STRING);
  BEGIN (* Asser *)
    IF (cond) THEN BEGIN
      WriteLn('ERROR: assertion failed - ', msg);
      HALT;
    END;
  END; (* Asser *)

  PROCEDURE InitVector;
  VAR
    i: INTEGER;
  BEGIN (* InitVector *)
    {$R-}
    FOR i := 1 TO Vsize DO BEGIN
      VPtr^[i] := 0;
    END; (* FOR *)
    {$R+}
  END; (* InitVector *)
```

```
  PROCEDURE DoubleSize;
  VAR
    newVPtr : ^Vector;
    i,j,prevSize: INTEGER;
  BEGIN (* DoubleSize *)
    prevSize := VSize;
    Vsize := Vsize*2;
    GetMem(newVPtr, Vsize * SizeOf(INTEGER));
    {$R-}
    FOR i := 1 TO Vsize DO BEGIN
      newVPtr^[i] := 0;
    END; (* FOR *)
    FOR j := 1 TO prevSize DO BEGIN
      newVPtr^[j] := VPtr^[j];
    END; (* FOR *)
    {$R+}
    FreeMem(VPtr, preVsize * SizeOf(INTEGER));
    GetMem(VPtr, Vsize * SizeOf(INTEGER));
    VPtr := newVPtr;
  END; (* DoubleSize *)

  PROCEDURE Add(val: INTEGER);
  VAR
    i: INTEGER;
  BEGIN
    i := 1;
    {$R-}
    WHILE (VPtr^[i] <> 0) AND (i <= Vsize) DO BEGIN
      Inc(i);
    END; (* WHILE *)
    {$R+}
    IF (i > Vsize) THEN BEGIN
      DoubleSize;
      {$R-}
      VPtr^[i] := val;
      {$R+}
    END ELSE BEGIN
      {$R-}
      VPtr^[i] := val;
      {$R+}
    END;
  END;

  PROCEDURE SetElementAt(pos: INTEGER; val: INTEGER);
```

```
BEGIN (* SetElementAt *)
  Assert((pos >= Vsize),'Stack Overflow');
  {$R-}
  VPtr^[pos] := val;
  {$R+}
END; (* SetElementAt *)

FUNCTION ElementAt(pos: INTEGER): INTEGER;
BEGIN (* ElementAt *)
  Assert((pos >= Vsize),'pos doesnt exist');
  ElementAt:= VPtr^[pos];
END; (* ElementAt *)

PROCEDURE RemoveElementAt(pos:INTEGER);
  VAR
    i: INTEGER;
BEGIN (* RemoveElementAt *)
  Assert((pos >= Vsize),'pos doesnt exist');
  {$R-}
  VPtr^[pos] := 0;
  FOR i := pos TO Vsize DO BEGIN
    VPtr^[i] := VPtr^[i+1];
  END; (* FOR *)
  VPtr^[VSize] := 0;
  {$R+}
END; (* RemoveElementAt *)

FUNCTION Size: INTEGER;
  VAR
    i,count: INTEGER;
BEGIN (* Size *)
  i := 1;
  count := 0;
  {$R-}
  WHILE (VPtr^[i] <> 0) AND (i <> Vsize+1) DO BEGIN
    Inc(count);
    Inc(i);
  END; (* WHILE *)
  {$R+}
  Size := count;
END; (* Size *)

FUNCTION Capacity: INTEGER;
BEGIN (* Capacity *)
  Capacity := Vsize;
```

```
  END; (* Capacity *)

  PROCEDURE WriteVector;
  VAR
    i: INTEGER;
  BEGIN (* WriteVector *)
    {$R-}
    FOR i := 1 TO Vsize DO BEGIN
      WriteLn(i,': ',VPtr^[i]);
    END; (* FOR *)
    {$R+}
  END; (* WriteVector *)


BEGIN
  Vsize := startSize;
  GetMem(VPtr, Vsize * SizeOf(INTEGER));
  InitVector;
  WriteLn('Loaded VectorADS');
END.
```

**VectorADSTest:**

```
PROGRAM VectorTest;

USES
  VectorADS;

VAR
  i: INTEGER;
BEGIN (* VectorTest *)
  WriteLn('Add 20 ints to Vector');
  FOR i := 1 TO 20 DO BEGIN
    Add(i);
  END; (* FOR *)
  WriteLn('SetElement at pos 11 to val 88');
  SetElementAt(11,88);
  WriteLn('RemoveElement At 12');
  RemoveElementAt(12);
  WriteLn('Writing Vector its Size and Capacity');
  WriteVector;
  WriteLn('Size: ',Size);
  WriteLn('Capacity: ',Capacity);

END. (* VectorTest *)
```

1.b) **Lösungsidee:** Viel ändert sich hierbei am code nicht man muss halt einpaar Sachen ändern damit als abstrakte Datentyp kompatibel ist.

1.b.2) **Testfälle**

Testfall1)

```
Loaded VectorADS
Adding 20 ints to v1[Cap = 10]
Adding 5 ints to v2[Cap = 5]
values in v1
1: 1
2: 2
3: 3
4: 4
5: 5
6: 6
7: 7
8: 8
9: 9
10: 10
11: 11
12: 12
13: 13
14: 14
15: 15
16: 16
17: 17
18: 18
19: 19
20: 20
values in v2
1: 1
2: 2
3: 3
4: 4
5: 5
```

Testfall2)

```
Loaded VectorADS
Adding 8 ints to v1[Cap = 5]
Adding 5 ints to v2[Cap = 5]
SetElement at v1 at pos 5 to val 88
Remove Element at v1 at pos 1
values in v1
1: 2
2: 3
3: 4
4: 88
5: 6
6: 7
7: 8
8: 0
9: 0
10: 0
values in v2
1: 1
2: 2
3: 3
4: 4
5: 5
Size of v1
7
Capacity of v1
10
Size of v2
4
Capacity of v2
5
```

1.b.3) **Quellcode**

**UNIT:**

```
UNIT VectorADT;

INTERFACE

  TYPE
    Vector = Pointer;

  PROCEDURE Assert(cond: BOOLEAN; msg: STRING);
  FUNCTION NewVector(size: INTEGER): Vector;
  PROCEDURE InitVector(VAR v: Vector);
  PROCEDURE DoubleSize(VAR v: Vector);
  PROCEDURE Add(VAR v: Vector; val: INTEGER);
  PROCEDURE SetElementAt(v: Vector; pos: INTEGER; val: INTEGER);
```

```
  FUNCTION ElementAt(v: Vector; pos: INTEGER): INTEGER;
  PROCEDURE RemoveElementAt(v: Vector; pos: INTEGER);
  FUNCTION Size(v: Vector): INTEGER;
  FUNCTION Capacity(v: Vector): INTEGER;
  PROCEDURE WriteVector(v: Vector);


IMPLEMENTATION

  TYPE
    VPtr = ^VRec;
    VRec = RECORD
      Vsize: INTEGER;
      data: ARRAY [1..1] OF INTEGER;
    END;

  PROCEDURE Assert(cond: BOOLEAN;msg: STRING);
  BEGIN (* Asser *)
    IF (cond) THEN BEGIN
      WriteLn('ERROR: assertion failed - ', msg);
      HALT;
    END;
  END; (* Asser *)

  FUNCTION NewVector(size: INTEGER): Vector;
    VAR
      v : VPtr;
  BEGIN (* NewVector *)
    GetMem(v,(size+1) * SizeOf(INTEGER));
    VPtr(v)^.Vsize := size;
    InitVector(v);
    NewVector := v;
  END; (* NewVector *)

  PROCEDURE InitVector(VAR v: Vector);
  VAR
    i: INTEGER;
  BEGIN (* InitVector *)
    {$R-}
    FOR i := 1 TO VPtr(v)^.Vsize DO BEGIN
      VPtr(v)^.data[i] := 0;
    END; (* FOR *)
    {$R+}
  END; (* InitVector *)
```

```
PROCEDURE DoubleSize(VAR v: Vector);
VAR
  newVPtr : Vector;
  i,j,prevSize,currentSize: INTEGER;
BEGIN (* DoubleSize *)
  prevSize := VPtr(v)^.Vsize;
  currentSize := prevSize * 2;
  newVPtr := NewVector(currentSize);
  {$R-}
  FOR i := 1 TO currentSize DO BEGIN
    VPtr(newVPtr)^.data[i] := 0;
  END; (* FOR *)
  FOR j := 1 TO prevSize DO BEGIN
    VPtr(newVPtr)^.data[j] := VPtr(v)^.data[j];
  END; (* FOR *)
  {$R+}
  FreeMem(v, (preVsize+1) * SizeOf(INTEGER));
  GetMem(v, ((prevSize*2)+1) * SizeOf(INTEGER));
  v := newVPtr;
  //WriteLn(VPtr(v)^.Vsize);
END; (* DoubleSize *)

PROCEDURE Add(VAR v: Vector;val: INTEGER);
VAR
  i: INTEGER;
BEGIN
  i := 1;
  //WriteLn(VPtr(v)^.Vsize,' start ',val);
  {$R-}
  WHILE (VPtr(v)^.data[i] <> 0) AND (i <= VPtr(v)^.Vsize) DO BEGIN
    //WriteLn(VPtr(v)^.Vsize,' start ',val);
    Inc(i);
  END; (* WHILE *)
  {$R+}
  IF (i > VPtr(v)^.Vsize) THEN BEGIN
    //WriteLn('here: ',i);
    DoubleSize(v);
    //WriteLn(VPtr(v)^.Vsize);
    {$R-}
    VPtr(v)^.data[i] := val;
    {$R+}
  END ELSE BEGIN
    {$R-}
    VPtr(v)^.data[i] := val;
    {$R+}
```

```
    END;
    //WriteLn(VPtr(v)^.Vsize);
  END;

  PROCEDURE SetElementAt(v: Vector;pos: INTEGER; val: INTEGER);
  BEGIN (* SetElementAt *)
    Assert((pos >= VPtr(v)^.Vsize),'Stack Overflow');
    {$R-}
    VPtr(v)^.data[pos] := val;
    {$R+}
  END; (* SetElementAt *)

  FUNCTION ElementAt(v: Vector;pos: INTEGER): INTEGER;
  BEGIN (* ElementAt *)
    Assert((pos >= VPtr(v)^.Vsize),'pos doesnt exist');
    ElementAt:= VPtr(v)^.data[pos];
  END; (* ElementAt *)

  PROCEDURE RemoveElementAt(v: Vector;pos:INTEGER);
    VAR
      i: INTEGER;
  BEGIN (* RemoveElementAt *)
    Assert((pos >= VPtr(v)^.Vsize),'pos doesnt exist');
    {$R-}
    VPtr(v)^.data[pos] := 0;
    FOR i := pos TO VPtr(v)^.Vsize DO BEGIN
      VPtr(v)^.data[i] := VPtr(v)^.data[i+1];
    END; (* FOR *)
    VPtr(v)^.data[VPtr(v)^.VSize] := 0;
    {$R+}
  END; (* RemoveElementAt *)

  FUNCTION Size(v: Vector): INTEGER;
    VAR
      i,count: INTEGER;
  BEGIN (* Size *)
    i := 1;
    count := 0;
    {$R-}
    WHILE (VPtr(v)^.data[i] <> 0) AND (i <> VPtr(v)^.Vsize)DO BEGIN
      Inc(count);
      Inc(i);
    END; (* WHILE *)
    {$R+}
    Size := count;
```

```
  END; (* Size *)

  FUNCTION Capacity(v: Vector): INTEGER;
  BEGIN (* Capacity *)
    Capacity := VPtr(v)^.Vsize;
  END; (* Capacity *)

  PROCEDURE WriteVector(v: Vector);
  VAR
    i: INTEGER;
  BEGIN (* WriteVector *)
    {$R-}
    FOR i := 1 TO VPtr(v)^.Vsize DO BEGIN
      WriteLn(i,': ',VPtr(v)^.data[i]);
    END; (* FOR *)
    {$R+}
  END; (* WriteVector *)


BEGIN
  WriteLn('Loaded VectorADS');
END.
```

**VectrorADTTest:**

```
PROGRAM VectorTest;

USES
  VectorADT;

VAR
  i,j: INTEGER;
  v1,v2: Vector;
BEGIN (* VectorTest *)
  v1 := NewVector(5);
  v2 := NewVector(5);
  WriteLn('Adding 8 ints to v1[Cap = 5]');
  FOR i := 1 TO 8 DO BEGIN
    Add(v1,i);
  END; (* FOR *)
  WriteLn('Adding 5 ints to v2[Cap = 5]');
  FOR j := 1 TO 5 DO BEGIN
    Add(v2,j);
  END; (* FOR *)
  WriteLn('SetElement at v1 at pos 5 to val 88');
```

```
  SetElementAt(v1,5,88);
  WriteLn('Remove Element at v1 at pos 1');
  RemoveElementAt(v1,1);
  WriteLn('values in v1');
  WriteVector(v1);
  WriteLn('values in v2');
  WriteVector(v2);
  WriteLn('Size of v1');
  WriteLn(Size(v1));
  WriteLn('Capacity of v1');
  WriteLn(Capacity(v1));
  WriteLn('Size of v2');
  WriteLn(Size(v2));
  WriteLn('Capacity of v2');
  WriteLn(Capacity(v2));

END. (* VectorTest *)
```

2) **Lösungsidee**: Das Queue Programm ist ähnlich wie das in der Übung bearbeitete Stack-Programm. Anstatt aber dass es Last in First out ist, ist eine Queue First in First out. Hierbei muss man beachten, dass beim dequeuen alles Elemente verschoben werden müssen. Abgesehen davon ist es das gleiche Prinzip wie das Stack Programm. Ebenfalls wird die Größe der Queue wie bei der ersten Aufgabe dynamisch festgelegt.

**Testfälle**:

Testfall1)

```
Unit QueueADS Loaded
Is Queue Empty?: TRUE
Filling queue with 20 Numbers
Dequeing 1 Number
Writing Queue
1: 19
2: 18
3: 17
4: 16
5: 15
6: 14
7: 13
8: 12
9: 11
10: 10
11: 9
12: 8
13: 7
14: 6
15: 5
16: 4
17: 3
18: 2
19: 1
20: 0
Is Queue Empty?: FALSE
```

**Quellcode**:

**UNIT**:

```
UNIT QueueADS;

INTERFACE
  PROCEDURE InitQueue;
  PROCEDURE DoubleSize;
  FUNCTION IsEmpty : BOOLEAN;
  PROCEDURE Enque(val: INTEGER);
  FUNCTION GetLast: INTEGER;
  PROCEDURE Deque;
  PROCEDURE WriteQueue;

IMPLEMENTATION

  TYPE
    Queue = ARRAY [1..1] OF INTEGER;
  VAR
    size: INTEGER;
    QPtr: ^Queue;

  PROCEDURE InitQueue;
    VAR
      i: INTEGER;
  BEGIN (* InitQueue *)
    {$R-}
    FOR i := 1 TO size DO BEGIN
      QPtr^[i] := 0;
    END; (* FOR *)
    {$R+}
  END; (* InitQueue *)

  PROCEDURE DoubleSize;
  VAR
    newQPtr : ^Queue;
    i,j,prevSize: INTEGER;
  BEGIN (* DoubleSize *)
    prevSize := size;
    size := size*2;
    GetMem(newQPtr, size * SizeOf(INTEGER));
    {$R-}
    FOR i := 1 TO size DO BEGIN
      newQPtr^[i] := 0;
```

```
    END; (* FOR *)
    FOR j := 1 TO prevSize DO BEGIN
      newQPtr^[j] := QPtr^[j];
    END; (* FOR *)
    {$R+}
    FreeMem(QPtr, prevsize * SizeOf(INTEGER));
    GetMem(QPtr, size * SizeOf(INTEGER));
    QPtr := newQPtr;
  END; (* DoubleSize *)

  FUNCTION IsEmpty: BOOLEAN;
  BEGIN (* IsEmpty *)
    {$R-}
    IsEmpty := (QPtr^[1] = 0);
    {$R+}
  END; (* IsEmpty *)

  PROCEDURE Enque(val: INTEGER);
    VAR
      i,prev,temp: INTEGER;
  BEGIN (* Enque *)
    {$R-}
    prev := QPtr^[1];
    i := 2;
    WHILE (QPtr^[i] <> 0) AND (i <> size) DO BEGIN
      temp := QPtr^[i];
      QPtr^[i] := prev;
      prev := temp;
      Inc(i);
    END; (* WHILE *)
    IF (QPtr^[i] = 0) THEN BEGIN
      temp := QPtr^[i];
      QPtr^[i] := prev;
      prev := temp;
    END;
    {$R+}
    IF (i = size) THEN BEGIN
      DoubleSize;
    END; (* IF *)
    {$R-}
    QPtr^[1] := val;
    {$R+}
  END; (* Enque *)

  FUNCTION GetLast: INTEGER;
```

```
    VAR
      i : INTEGER;
  BEGIN (* GetLast *)
    i := 1;
    {$R-}
    WHILE QPtr^[i] <> 0 DO BEGIN
      Inc(i);
    END;
    {$R+}
    GetLast := i;
  END; (* GetLast *)

  PROCEDURE Deque;
    VAR
      i,prev,temp: INTEGER;
  BEGIN (* Deque *)
    i := size-1;
    {$R-}
    prev := QPtr^[size];
    QPtr^[1] := 0;
    WHILE (i <> 1) DO BEGIN
      temp := QPtr^[i];
      QPtr^[i] := prev;
      prev := temp;
      Dec(i);
    END; (* WHILE *)
    QPtr^[1] := prev;
    {$R+}
  END; (* Deque *)

  PROCEDURE WriteQueue;
    VAR
      i : INTEGER;
  BEGIN (* WriteQueue *)
    {$R-}
    FOR i := 1 TO 20 DO BEGIN
      WriteLn(i,': ',QPtr^[i]);
    END; (* FOR *)
    {$R+}
  END; (* WriteQueue *)

BEGIN
size := 10;
GetMem(QPtr,size*SizeOf(INTEGER));
InitQueue;
```

```
WriteLn('Unit QueueADS Loaded');
END.
```

**QUEUETEST**:

```
PROGRAM QueueTest;

USES QueueADS;


VAR
  i : INTEGER;
BEGIN (* QueueTest *)
  WriteLn('Is Queue Empty?: ',IsEmpty);
  WriteLn('Filling queue with 20 Numbers');
  FOR i := 1 TO 20 DO BEGIN
    Enque(i);
  END; (* FOR *)
  WriteLn('Dequeing 1 Number');
  Deque;
  WriteLn('Writing Queue');
  WriteQueue;
  WriteLn('Is Queue Empty?: ',IsEmpty);
END. (* QueueTest *)
```