Abgabetermin: Mi, 19. 04. 2023

☒ **Gr. 1,** Dr. S. Wagner **Name** Elias Wurm **Aufwand in h** ~3h 30min
☐ **Gr. 2,** Dr. D. Auer
☐ **Gr. 3,** Dr. G. Kronberger **Punkte** _____ **Kurzzeichen Tutor / Übungsleiter*in** _____ / _____

1. „Behälter“ *Vector* als ADT (12 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.

Implementieren Sie einen Behälter `Vector` für Elemente des Typs `INTEGER` als abstrakten Datentyp (in Form eines Moduls), der mindestens folgende Operationen bietet:

`PROCEDURE InitVector(VAR v: Vector);`
initialisiert ein neues `Vector`-Datenobjekt.

`PROCEDURE DisposeVector(VAR v: Vector);`
gibt den Speicherbereich eines `Vector`-Datenobjekts frei.

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

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

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

`PROCEDURE RemoveElementAt(VAR v: Vector; 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(v: Vector): INTEGER;`
liefert die aktuelle Anzahl der im Behälter gespeicherten Werte (zu Beginn 0).

`FUNCTION Capacity(v: Vector): 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.

2. ADT: Multiset (12 Punkte)

Ein *Multiset* (oder *bag*) im mathematischen Sinn ist eine Menge, in der gleiche Elemente auch mehrfach enthalten sein können. Ein *Multiset* speichert also nicht nur, ob ein Element enthalten ist, sondern auch, wie oft dieses Element vorkommt.

Implementieren Sie ein *Multiset* für Zeichenketten (Datentyp `STRING`) auf Basis eines binären Suchbaums als abstrakten Datentyp in Form eines Moduls (Pascal-*UNIT*). Für den abstrakten Datentyp z.B. mit dem Bezeichner `StrMSet` müssen mindestens folgende Operationen angeboten werden:

```
PROCEDURE InitStrMSet(VAR ms: StrMSet);
PROCEDURE DisposeStrMSet(VAR ms: StrMSet);
PROCEDURE Insert(VAR ms: StrMSet; element: STRING);
PROCEDURE Remove(VAR ms: StrMSet; element: STRING);
FUNCTION  IsEmpty(ms: StrMSet): BOOLEAN;
FUNCTION  Contains(ms: StrMSet; element: STRING): BOOLEAN;
FUNCTION  Count(ms: StrMSet; element: STRING): INTEGER;
FUNCTION  Cardinality(ms: StrMSet): INTEGER;
FUNCTION  CountUnique(ms: StrMSet): INTEGER;
```

Hinweis: die Operation `Cardinality` liefert die Anzahl aller Elemente in einer *Multiset*-Datenstruktur, die Operation `CountUnique` liefert die Anzahl verschiedener Elemente.

Implementieren Sie ein Pascal-Hauptprogramm, um Ihre *Multiset*-Implementierung zu testen.

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.

Aufgabe 1 - „Behälter“ Vector als ADT

Lösungsidee:
wie bei dem beispiel wo die größe des Felds erst zur Laufzeit fixiert wird ein array auf die gleiche Weise erstellen allerdings immer wenn die größe überschritten werden sollte, wird die funktion GrowVector aufgerufen, die die capacity verdopellt und somit auch den allokierten speicher und die werte des allten vectors in den neuen kopiert.

eine mögliche verbesserung meines codes wäre den vector wieder zu verkleinern wenn der count kleiner als die hälfte von capacity ist, oder den vector nicht jedesmal ums doppelte zu vergrößern sondern eine bessere gewählten wert zu verwenden. Aber das ist immer usecase spezifisch, je nachdem was man mit dem vector anfangen will.

Zeitaufwand: ~1h 45min

Code:

```
unit VectorUnit;

interface

type
  Vector = Pointer;

procedure InitVector(var v: Vector);
procedure DisposeVector(var v: Vector);
procedure Add(var v: Vector; val: Integer);
procedure SetElementAt(var v: Vector; pos: Integer; val: Integer);
function ElementAt(v: Vector; pos: Integer): Integer;
procedure RemoveElementAt(var v: Vector; pos: Integer);
function Size(v: Vector): Integer;
function Capacity(v: Vector): Integer;

implementation

// using MaxInt for the intArray here so i dont have to disable rangecheck
// error everytime i try to access the array if i used array[0..0] instead.
// There is no allocation of more memory as a consequence because i manually
// set the memory with capacity * SizeOf(Integer) so there shouldnt be a
// problem, also i choose MaxInt because my count and capacity are int anyway
type
  IntArray = array[0..MaxInt] of Integer;
  PIntArray = ^IntArray;
  VecRec = record
```

```
    data: PIntArray;
    count: Integer;
    capacity: Integer;
  end;
  PVector = ^VecRec;

procedure InitVector(var v: Vector);
var
  pv: PVector;
begin
  pv := PVector(v);
  pv^.count := 0;
  pv^.capacity := 1;
  GetMem(pv^.data, pv^.capacity * SizeOf(Integer));
  if pv^.data = nil then
  begin
    WriteLn('Error: Heap overflow.');
    Halt(1);
  end;
end;

procedure DisposeVector(var v: Vector);
var
  pv: PVector;
begin
  pv := PVector(v);
  FreeMem(pv^.data, pv^.capacity * SizeOf(Integer));
end;

procedure GrowVector(var v: VecRec);
var
  newCapacity: longInt;
  newData: PIntArray;
  i: Integer;
begin
  newCapacity := v.capacity * 2;
  if newCapacity >= MaxInt then
  begin
    WriteLn('Error: Vector overflow.');
    Halt(1);
  end;
  GetMem(newData, newCapacity * SizeOf(Integer));
  for i := 0 to v.count - 1 do
    newData^[i] := v.data^[i];
  FreeMem(v.data, v.capacity * SizeOf(Integer));
  v.data := newData;
  v.capacity := newCapacity;
end;
```

```
procedure Add(var v: Vector; val: Integer);
var
  pv: PVector;
begin
  pv := PVector(v);
  if pv^.count = pv^.capacity then
    GrowVector(pv^);
  pv^.data^[pv^.count] := val;
  Inc(pv^.count);
end;

procedure SetElementAt(var v: Vector; pos: Integer; val: Integer);
var
  pv: PVector;
begin
  pv := PVector(v);
  if (pos < 0) or (pos >= pv^.count) then
  begin
    WriteLn('Error: Index out of range.');
    Halt(1);
  end;
  pv^.data^[pos] := val;

end;

function ElementAt(v: Vector; pos: Integer): Integer;
var
  pv: PVector;
begin
  pv := PVector(v);
  if (pos < 0) or (pos >= pv^.count) then
  begin
    WriteLn('Error: Index out of range.');
    Halt(1);
  end;
  ElementAt := pv^.data^[pos];
end;

procedure RemoveElementAt(var v: Vector; pos: Integer);
var
  i: Integer;
  pv: PVector;
begin
  pv := PVector(v);
  if (pos < 0) or (pos >= pv^.count) then
  begin
    WriteLn('Error: Index out of range.');
```

```
    Halt(1);
  end;
  for i := pos to pv^.count - 2 do
    pv^.data^[i] := pv^.data^[i + 1];
  Dec(pv^.count);
end;

function Size(v: Vector): Integer;
var
  pv: PVector;
begin
  pv := PVector(v);
  Size := pv^.count;
end;

function Capacity(v: Vector): Integer;
var
  pv: PVector;
begin
  pv := PVector(v);
  Capacity := pv^.capacity;
end;

end.
```

Test Code:

```
program VectorTests;

uses VectorUnit;

var
  v: Vector;
begin
  // Initialize an empty vector
  InitVector(v);

  // Test adding elements
  WriteLn('Test adding elements:');
  writeln('0 elements - size: ', Size(v), ', capacity: ', Capacity(v));
  Add(v, 1);
  writeln('1 element - size: ', Size(v), ', capacity: ', Capacity(v));
  Add(v, 2);
  writeln('2 elements - size: ', Size(v), ', capacity: ', Capacity(v));
  Add(v, 3);
  writeln('3 elements - size: ', Size(v), ', capacity: ', Capacity(v));
```

```
  Add(v, 4);
  writeln('4 elements - size: ', Size(v), ', capacity: ', Capacity(v));
  Add(v, 5);
  writeln('5 elements - size: ', Size(v), ', capacity: ', Capacity(v));
  WriteLn;
  WriteLn;

  // Test getting and setting elements
  WriteLn('Test getting and setting elements:');
  WriteLn('Element at position 2: ', ElementAt(v, 2)); // should print 3
  SetElementAt(v, 2, 6);
  WriteLn('Element at position 2 after setting it to 6: ', ElementAt(v, 2));
// should print 6
  WriteLn;
  WriteLn;

  // Test removing elements
  WriteLn('Test removing elements:');
  WriteLn('Vector size before removing an element: ', Size(v)); // should
print 5
  RemoveElementAt(v, 2);
  WriteLn('Vector size after removing an element: ', Size(v)); // should print
4
  WriteLn('Element at position 2 after removing element at position 2: ',
ElementAt(v, 2)); // should print 4
  WriteLn;
  WriteLn;

  // Test disposing of vector
  DisposeVector(v);
end.
```

Test Ausgabe:

```
Test adding elements:
0 elements - size: 0, capacity: 1
1 element - size: 1, capacity: 1
2 elements - size: 2, capacity: 2
3 elements - size: 3, capacity: 4
4 elements - size: 4, capacity: 4
5 elements - size: 5, capacity: 8


Test getting and setting elements:
Element at position 2: 3
Element at position 2 after setting it to 6: 6


Test removing elements:
Vector size before removing an element: 5
Vector size after removing an element: 4
Element at position 2 after removing element at position 2: 4


Heap dump by heaptrc unit of C:\Repos\2023SS_ADF\UE5\hu\vector-tests.exe
103 memory blocks allocated : 2332/2600
103 memory blocks freed     : 2332/2600
0 unfreed memory blocks : 0
True heap size : 163840 (96 used in System startup)
True free heap : 163744
```

Aufgabe 1 - „Behälter“ Vector als ADT

Lösungsidee:
Wie bei dem Beispiel wo die Größe des Feldes erst zur Laufzeit fixiert wird ein Feld auf die gleiche Weise erstellen, allerdings immer wenn die Größe überschritten werden sollte, wird die Funktion GrowVector aufgerufen, die die Kapazität des Feldes verdoppelt und somit auch den allokierten Speicher und die Werte des alten Vektors in den neuen kopiert.

Zeitaufwand: ~1h 45min

Code:

```
unit VectorUnit;

interface

type
  Vector = Pointer;

procedure InitVector(var v: Vector);
procedure DisposeVector(var v: Vector);
procedure Add(var v: Vector; val: Integer);
procedure SetElementAt(var v: Vector; pos: Integer; val: Integer);
function ElementAt(v: Vector; pos: Integer): Integer;
procedure RemoveElementAt(var v: Vector; pos: Integer);
function Size(v: Vector): Integer;
function Capacity(v: Vector): Integer;

implementation

// using MaxInt for the intArray here so i dont have to disable rangecheck
// error everytime i try to access the array if i used array[0..0] instead.
// There is no allocation of more memory as a consequence because i manually
// set the memory with capacity * SizeOf(Integer) so there shouldnt be a
// problem, also i choose MaxInt because my count and capacity are int anyway
type
  IntArray = array[0..MaxInt] of Integer;
  PIntArray = ^IntArray;
  VecRec = record
    data: PIntArray;
    count: Integer;
    capacity: Integer;
  end;
  PVector = ^VecRec;

procedure InitVector(var v: Vector);
```

```
var
  pv: PVector;
begin
  new(pv);
  pv^.count := 0;
  pv^.capacity := 1;
  GetMem(pv^.data, pv^.capacity * SizeOf(Integer));
  if pv^.data = nil then
  begin
    WriteLn('Error: Heap overflow.');
    Halt(1);
  end;
  v := pv;
end;

procedure DisposeVector(var v: Vector);
var
  pv: PVector;
begin
  pv := PVector(v);
  FreeMem(pv^.data, pv^.capacity * SizeOf(Integer));
  Dispose(v);
end;

procedure GrowVector(var v: VecRec);
var
  newCapacity: longInt;
  newData: PIntArray;
  i: Integer;
begin
  newCapacity := v.capacity * 2;
  if newCapacity >= MaxInt then
  begin
    WriteLn('Error: Vector overflow.');
    Halt(1);
  end;
  GetMem(newData, newCapacity * SizeOf(Integer));
  for i := 0 to v.count - 1 do
    newData^[i] := v.data^[i];
  FreeMem(v.data, v.capacity * SizeOf(Integer));
  v.data := newData;
  v.capacity := newCapacity;
end;

procedure Add(var v: Vector; val: Integer);
var
  pv: PVector;
begin
```

```
  pv := PVector(v);
  if pv^.count = pv^.capacity then
    GrowVector(pv^);
  pv^.data^[pv^.count] := val;
  Inc(pv^.count);
end;

procedure SetElementAt(var v: Vector; pos: Integer; val: Integer);
var
  pv: PVector;
begin
  pv := PVector(v);
  if (pos < 0) or (pos >= pv^.count) then
  begin
    WriteLn('Error: Index out of range.');
    Halt(1);
  end;
  pv^.data^[pos] := val;

end;

function ElementAt(v: Vector; pos: Integer): Integer;
var
  pv: PVector;
begin
  pv := PVector(v);
  if (pos < 0) or (pos >= pv^.count) then
  begin
    WriteLn('Error: Index out of range.');
    Halt(1);
  end;
  ElementAt := pv^.data^[pos];
end;

procedure RemoveElementAt(var v: Vector; pos: Integer);
var
  i: Integer;
  pv: PVector;
begin
  pv := PVector(v);
  if (pos < 0) or (pos >= pv^.count) then
  begin
    WriteLn('Error: Index out of range.');
    Halt(1);
  end;
  for i := pos to pv^.count - 2 do
    pv^.data^[i] := pv^.data^[i + 1];
  Dec(pv^.count);
```

```
end;

function Size(v: Vector): Integer;
var
  pv: PVector;
begin
  pv := PVector(v);
  Size := pv^.count;
end;

function Capacity(v: Vector): Integer;
var
  pv: PVector;
begin
  pv := PVector(v);
  Capacity := pv^.capacity;
end;

end.
```

Eine mögliche Verbesserung meines Codes wäre den Vektor wieder zu verkleinern, wenn der Count kleiner als die Hälfte von der Kapazität ist, oder den Vektor nicht jedes Mal ums doppelte zu vergrößern sondern eine bessere gewählten wert zu verwenden. Aber das ist immer usecase spezifisch, je nachdem, was man mit dem Vektor anfangen will.

Test Code:

```
program VectorTests;

uses VectorUnit;

var
  v: Vector;
begin
  // Initialize an empty vector
  InitVector(v);

  // Test adding elements
  WriteLn('Test adding elements:');
  writeln('0 elements - size: ', Size(v), ', capacity: ', Capacity(v));
  Add(v, 1);
  writeln('1 element - size: ', Size(v), ', capacity: ', Capacity(v));
  Add(v, 2);
  writeln('2 elements - size: ', Size(v), ', capacity: ', Capacity(v));
  Add(v, 3);
```

```
  writeln('3 elements - size: ', Size(v), ', capacity: ', Capacity(v));
  Add(v, 4);
  writeln('4 elements - size: ', Size(v), ', capacity: ', Capacity(v));
  Add(v, 5);
  writeln('5 elements - size: ', Size(v), ', capacity: ', Capacity(v));
  WriteLn;
  WriteLn;

  // Test getting and setting elements
  WriteLn('Test getting and setting elements:');
  WriteLn('Element at position 2: ', ElementAt(v, 2)); // should print 3
  SetElementAt(v, 2, 6);
  WriteLn('Element at position 2 after setting it to 6: ', ElementAt(v, 2));
// should print 6
  WriteLn;
  WriteLn;

  // Test removing elements
  WriteLn('Test removing elements:');
  WriteLn('Vector size before removing an element: ', Size(v)); // should
print 5
  RemoveElementAt(v, 2);
  WriteLn('Vector size after removing an element: ', Size(v)); // should print
4
  WriteLn('Element at position 2 after removing element at position 2: ',
ElementAt(v, 2)); // should print 4
  WriteLn;
  WriteLn;

  // Test disposing of vector
  DisposeVector(v);
end.
```

Test Ausgabe:

```
Test adding elements:
0 elements - size: 0, capacity: 1
1 element - size: 1, capacity: 1
2 elements - size: 2, capacity: 2
3 elements - size: 3, capacity: 4
4 elements - size: 4, capacity: 4
5 elements - size: 5, capacity: 8


Test getting and setting elements:
Element at position 2: 3
Element at position 2 after setting it to 6: 6


Test removing elements:
Vector size before removing an element: 5
Vector size after removing an element: 4
Element at position 2 after removing element at position 2: 4


Heap dump by heaptrc unit of C:\Repos\2023SS_ADF\UE5\hu\vector-tests.exe
103 memory blocks allocated : 2332/2600
103 memory blocks freed     : 2332/2600
0 unfreed memory blocks : 0
True heap size : 163840 (96 used in System startup)
True free heap : 163744
```