

<input checked="" type="checkbox"/> Gr. 1, Dr. S. Wagner	Name <u>Elias Wurm</u>	Aufwand in h <u>~3h 30min</u>
<input type="checkbox"/> Gr. 2, Dr. D. Auer		
<input type="checkbox"/> 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 verdoppelt 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];
    end;
    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 2 - ADT: Multiset

Lösungsidee:

Standardfunktionen eines BST einbauen für strings und helper Funktionen/Prozeduren verwenden, um so viel code Duplikation wie möglich zu vermeiden.

Zeitaufwand: ~1h 45min

Code:

```
unit Multiset;

interface

type
    StrMSet = pointer;

procedure InitStrMSet(var ms: StrMSet);
procedure DisposeStrMSet(var ms: StrMSet);
procedure Insert(var ms: StrMSet; value: STRING);
procedure Remove(var ms: StrMSet; value: STRING);
function IsEmpty(ms: StrMSet): BOOLEAN;
function Contains(ms: StrMSet; value: STRING): BOOLEAN;
function Count(ms: StrMSet; value: STRING): INTEGER;
function Cardinality(ms: StrMSet): INTEGER;
function CountUnique(ms: StrMSet): INTEGER;
// helper procedure for easier debugging
procedure PrintTree(ms: StrMSet);

implementation

type
    PstrNode = ^StrNode;
    StrNode = record
        value: string;
        count: integer;
        left: PstrNode;
        right: PstrNode;
    end;
    PMSet = ^MSet;
    MSet = record
        root: PstrNode;
    end;

//region: helper functions
function NewStrNode(value: string): PstrNode;
```

```

var
    node: PstrNode;
begin
    New(node);
    node^.value := value;
    node^.count := 1;
    node^.left := nil;
    node^.right := nil;
    NewStrNode := node;
end;

function InsertStrNode(var node: PstrNode; value: string): PstrNode;
begin
    if node = nil then
    begin
        InsertStrNode := NewStrNode(value);
        Exit;
    end;

    if value < node^.value then
        node^.left := InsertStrNode(node^.left, value)
    else if value > node^.value then
        node^.right := InsertStrNode(node^.right, value)
    else
        Inc(node^.count);

    InsertStrNode := node;
end;

function RemoveStrNode(var node: PstrNode; value: string): PstrNode;
var
    successor: PstrNode;
begin
    if node = nil then
    begin
        RemoveStrNode := nil;
        Exit;
    end;

    if value < node^.value then
        node^.left := RemoveStrNode(node^.left, value)
    else if value > node^.value then
        node^.right := RemoveStrNode(node^.right, value)
    else if node^.count > 1 then
        Dec(node^.count)
    else if node^.left = nil then
    begin
        RemoveStrNode := node^.right;

```

```

    Dispose(node);
    Exit;
end else if node^.right = nil then
begin
    RemoveStrNode := node^.left;
    Dispose(node);
    Exit;
end else begin
    successor := node^.right;
    while successor^.left <> nil do
        successor := successor^.left;
    end;
    node^.value := successor^.value;
    node^.count := successor^.count;
    node^.right := RemoveStrNode(node^.right, successor^.value);
end;

RemoveStrNode := node;
end;

function FindStrNode(node: PStrNode; value: STRING): PStrNode;
begin
    if node = nil then
        FindStrNode := nil
    else if value < node^.value then
        FindStrNode := FindStrNode(node^.left, value)
    else if value > node^.value then
        FindStrNode := FindStrNode(node^.right, value)
    else
        FindStrNode := node;
    end;
end;

function CountNodeValues(node: PstrNode; uniqueOnly: Boolean): Integer;
begin
    if node = nil then
        CountNodeValues := 0
    else if uniqueOnly then
        CountNodeValues := 1 + CountNodeValues(node^.left, uniqueOnly) +
        CountNodeValues(node^.right, uniqueOnly)
    else
        CountNodeValues := node^.count + CountNodeValues(node^.left, uniqueOnly) +
        CountNodeValues(node^.right, uniqueOnly);
    end;
end;

procedure DisposeStrNode(node: PStrNode);
begin
    if node <> nil then
        begin
            DisposeStrNode(node^.left);

```

```

        DisposeStrNode(node^.right);
        Dispose(node);
    end;
end;

procedure PrintTreeNodes(root: PstrNode; level: integer);
var
    i: integer;
begin
    if root = nil then
        exit;

    PrintTreeNodes(root^.right, level + 1);

    for i := 1 to level do
        write(' ');

    writeln(root^.value, ': ', root^.count);

    PrintTreeNodes(root^.left, level + 1);
end;

//regionEnd helper functions

procedure InitStrMSet(var ms: StrMSet);
var
    pms: PMSet;
begin
    if ms = nil then
        begin
            New(pms);
            pms^.root := nil;
            ms := pms;
        end else begin
            pms := PMSet(ms);
            pms^.root := nil;
        end;
end;

procedure DisposeStrMSet(var ms: StrMSet);
var
    pms: PMSet;
begin
    pms := PMSet(ms);
    if pms^.root <> nil then
        DisposeStrNode(pms^.root);
    pms^.root := nil;
    Dispose(pms);
end;

```

```

end;

procedure Insert(var ms: StrMSet; value: string);
var
    pms: PMSet;
begin
    pms := PMSet(ms);
    pms^.root := InsertStrNode(pms^.root, value);
end;

procedure Remove(var ms: StrMSet; value: string);
var
    pms: PMSet;
begin
    pms := PMSet(ms);
    pms^.root := RemoveStrNode(pms^.root, value);
end;

function IsEmpty(ms: StrMSet): Boolean;
var
    pms: PMSet;
begin
    pms := PMSet(ms);
    IsEmpty := pms^.root = nil;
end;

function Contains(ms: StrMSet; value: STRING): BOOLEAN;
var
    pms: PMSet;
begin
    pms := PMSet(ms);
    Contains := FindStrNode(pms^.root, value) <> nil;
end;

function Count(ms: StrMSet; value: STRING): INTEGER;
var
    node: PStrNode;
    pms: PMSet;
begin
    pms := PMSet(ms);
    node := FindStrNode(pms^.root, value);
    if node <> nil then
        Count := node^.count
    else
        Count := 0;
    end;
end;

function Cardinality(ms: StrMSet): Integer;

```

```

var
  pms: PMSet;
begin
  pms := PMSet(ms);
  Cardinality := CountNodeValues(pms^.root, False);
end;

function CountUnique(ms: StrMSet): Integer;
var
  pms: PMSet;
begin
  pms := PMSet(ms);
  CountUnique := CountNodeValues(pms^.root, True);
end;

procedure PrintTree(ms: StrMSet);
var
  pms: PMSet;
begin
  pms := PMSet(ms);
  PrintTreeNodes(pms^.root, 0);
end;

end.

```

Test Code:

```

program StrMSetTests;

uses Multiset;

var
  ms: StrMSet;
begin
  // Initialize an empty multiset
  InitStrMSet(ms);

  // Test inserting elements
  Insert(ms, 'apple');
  Insert(ms, 'banana');
  Insert(ms, 'apple');
  Insert(ms, 'cherry');
  Insert(ms, 'banana');
  Insert(ms, 'cherry');

```

```

// Test counting elements
WriteLn('Test counting elements:');
WriteLn('Count of 'apple': ', Count(ms, 'apple')); // should print 2
WriteLn('Count of 'banana': ', Count(ms, 'banana')); // should print 2
WriteLn('Count of 'cherry': ', Count(ms, 'cherry')); // should print 2
WriteLn('Count of 'grape': ', Count(ms, 'grape')); // should print 0
WriteLn;
WriteLn;

// Test count unique and cardinality
WriteLn('Test count unique and cardinality:');
WriteLn('Number of unique elements: ', CountUnique(ms)); // should print 3
WriteLn('Cardinality: ', Cardinality(ms)); // should print 6
WriteLn;
WriteLn;

// Test checking if elements are present
WriteLn('Test checking if elements are present');
WriteLn('Contains 'apple': ', Contains(ms, 'apple')); // should print True
WriteLn('Contains 'banana': ', Contains(ms, 'banana')); // should print
True
WriteLn('Contains 'cherry': ', Contains(ms, 'cherry')); // should print
True
WriteLn('Contains 'grape': ', Contains(ms, 'grape')); // should print
False
WriteLn;
WriteLn;

// Test removing elements
WriteLn('Test removing elements');
Remove(ms, 'apple');
WriteLn('Count of 'apple': ', Count(ms, 'apple')); // should print 1
Remove(ms, 'banana');
WriteLn('Count of 'banana': ', Count(ms, 'banana')); // should print 1
Remove(ms, 'cherry');
WriteLn('Count of 'cherry': ', Count(ms, 'cherry')); // should print 1
Remove(ms, 'cherry');
WriteLn('Count of 'cherry': ', Count(ms, 'cherry')); // should print 0
WriteLn;
WriteLn;

// Test checking if multiset is empty
WriteLn('Is empty: ', IsEmpty(ms)); // should print False

// Test disposing of multiset
DisposeStrMSet(ms);
WriteLn('Is empty: ', IsEmpty(ms)); // should print True
end.

```

Test Ausgabe:

› Test counting elements:

Count of 'apple': 2

Count of 'banana': 2

Count of 'cherry': 2

Count of 'grape': 0

Test count unique and cardinality:

Number of unique elements: 3

Cardinalities: 6

Test checking if elements are present

Contains 'apple': TRUE

Contains 'banana': TRUE

Contains 'cherry': TRUE

Contains 'grape': FALSE

Test removing elements

Count of 'apple': 1

Count of 'banana': 1

Count of 'cherry': 1

Count of 'cherry': 0

Is empty: FALSE

Is empty: TRUE