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.