ADF 2*x* & PRO 2*x* **Übungen zu Fortgeschrittenen Algorithmen & Datenstrukturen und OOP** **SS 21, Übung 7**

Abgabetermin: Mi, 09. 06. 2021

☐ **Gr. 1,** Dr. S. Wagner **Name** Angelos Angelis **Aufwand in h** 8

☒ **Gr. 2,** Dr. D. Auer

☐ **Gr. 3,** Dr. G. Kronberger **Punkte** _____ **Kurzzeichen Tutor / Übungsleiter*in** _____ / _____

1. „Behälter“ *Vector* als Klasse (6 Punkte)

In Übung 4 haben wir bereits eine Abstrakte Datenstruktur (ADS) sowie einen Abstrakten Datentyp (ADT) für den Behälter *Vector* implementiert. Jetzt folgt die objektorientierte Version.

Entwickeln Sie eine Klasse *Vector* und testen Sie diese ausführlich.

Folgende Methoden muss die Klasse *Vector* mindestens bieten.

`PROCEDURE` **`Add`**`(val: INTEGER)`
fügt den Wert *val* „hinten“ an.

`PROCEDURE` **`InsertElementAt`**`(pos: INTEGER; val: INTEGER; VAR ok: BOOLEAN)`
fügt den Wert *val* an der Stelle *pos* ein, wobei die Werte ab dieser Stelle nach hinten verschoben werden. Ist *pos* ≤ 0, wird *val* „vorne“ eingefügt; ist *pos* > *Size* (s. u.) wird *val* „hinten“ angefügt. Der Ausgangsparameter *ok* liefert nur dann *FALSE*, wenn für *pos* ein Wert über der Obergrenze des Vektors (*Capacity*) angegeben wurde.

`PROCEDURE` **`GetElementAt`**`(pos: INTEGER; VAR val: INTEGER; VAR ok: BOOLEAN)`
liefert den Wert *val* an der Stelle *pos*. Der Ausgangsparameter *ok* liefert nur dann *FALSE*, wenn für *pos* ein ungültiger Wert angegeben wurde.

`FUNCTION` **`Size`**`: INTEGER`
liefert die aktuelle Anzahl der Elemente.

`PROCEDURE` **`Clear`**
leert den Behälter (Size liefert dann 0).

Sie können die Größe des Vektors als vorgegeben (über Konstante) annehmen. Wenn Sie möchten, können Sie die Klasse *Vector* aber auch so implementieren, dass erst beim Erstellen eines *Vector*-Objekts die Größe des Vektors festgelegt wird. Implementieren Sie die folgende Methode entsprechend ihres Lösungsansatzes.

`FUNCTION` **`Capacity`**`: INTEGER`
liefert die aktuelle Kapazität des Behälters (= max. Anzahl der Elemente, die der Vektor aufnehmen kann.

2. Klassen für Zeichenketten-Operationen (3 + 3 + 3 Punkte)

(a) Entwickeln Sie eine Klasse *StringBuilder* mit einer Datenkomponente *buffer* vom Typ *STRING* und (mindestens) folgenden Methoden:

`PROCEDURE` **`AppendStr`**`(e: STRING)`
fügt die Zeichenkette *e* an die Datenkomponente *buffer* an

`PROCEDURE` **`AppendChar`**`(e: CHAR)`
fügt das Zeichen *e* an *buffer* an

`PROCEDURE` **`AppendInt`**`(e: INTEGER)`
konvertiert den Wert von *e* (z.B. mit *Str*) in eine Zeichenkette und fügt diese an *buffer* an

`PROCEDURE` **`AppendBool`**`(e: BOOLEAN)`
fügt die Zeichenketten `'TRUE'` oder `'FALSE'` an die Datenkomponente *buffer* an

`FUNCTION` **`AsString:`** `STRING`
liefert den Inhalt der Datenkomponente *buffer*

(b) Entwickeln Sie eine von der Klasse *StringBuilder* abgeleitete Klasse *TabStringBuilder* und überschreiben Sie die Methoden, um angefügte Elemente *spaltenweise* (durch Auffüllen mit Leerzeichen) auszurichten. Die Spaltenbreite wird beim Erstellen eines *TabStringBuilder*-Objekts festgelegt.

(c) Entwickeln Sie eine Klasse *StringJoiner*, um aus mehreren Zeichenketten und einem Trennzeichen *delimiter* eine Zeichenkette zu bilden. Verwenden Sie für die Verkettung der Zeichenketten und Trennzeichen eine Datenkomponente vom Typ *StringBuilder* und implementieren Sie (mindestens) folgende Konstruktoren und Methoden:

`CONSTRUCTOR` **`Init`**`(delimiter: CHAR)`
initialisiert ein *StringJoiner*-Objekt mit dem Trennzeichen *delimiter*

`PROCEDURE` **`Add`**`(e: STRING)`
fügt die Zeichenkette *e* getrennt durch ein Trennzeichen an

`FUNCTION` **`AsString:`** `STRING`
liefert das Ergebnis als Zeichenkette

Beispiele für die Verwendung der Klassen *StringBuilder*, *TabStringBuilder* und *StringJoiner*:

```
VAR
  s: StringBuilder;
BEGIN
  New(s, Init);
  s^.AppendStr('Eins');
  s^.AppendChar(' ');
  s^.AppendInt(2);
  s^.AppendChar(' ');
  s^.AppendBool(TRUE);
  WriteLn(s^.AsString);
  …
```

Ausgabe:

```
Eins 2 TRUE
```

```
VAR
  t: TabStringBuilder;
BEGIN
  New(t, Init(8));
  t^.AppendStr('Eins');
  t^.AppendInt(2);
  t^.AppendBool(TRUE);
  WriteLn(t^.AsString);
  …
```

Ausgabe:

```
Eins    2       TRUE
```

```
VAR
  j: StringJoiner;
BEGIN
  New(j, Init(','));
  j^.Add('Eins');
  j^.Add('Zwei');
  j^.Add('Drei');
  WriteLn(j^.AsString);
  …
```

Ausgabe:

```
Eins,Zwei,Drei
```

3. Dateisystem als Klassen (9 Punkte)

Entwickeln Sie die erforderlichen Klassen, um ein Dateisystem (*File* und *Folder*) zu beschreiben. Beachten Sie, dass ein Verzeichnis (*Folder*) eine Aggregation aus „beliebig vielen“ Dateien (*File*) und Verzeichnissen (*Folder*) ist. Wird ein übergeordnetes Verzeichnis gelöscht / verschoben, so werden auch alle darin enthaltenen Elemente gelöscht / verschoben.

Die Klassen *File* und *Folder* beschreiben die Eigenschaften: *name* (*STRING*), *type* (*STRING*) und *dateModified* (*STRING*). *File* verfügt darüber hinaus auch über die Eigenschaft *size* (*LONGINT*), während ein *Folder*-Objekt in der Lage sein muss, „beliebig viele“ *File*- und *Folder*-Objekte aufzunehmen.

Entwickeln Sie alle notwendigen Methoden, um die grundlegenden Eigenschaften der *File*- und *Folder*-Objekte zu verwalten und diese Eigenschaften in eine Zeichenkette (*AsString*) zu schreiben.

Darüber hinaus sind für die Klasse *Folder* folgende Methoden gefordert:

`PROCEDURE` **`Add`**`(…)`
: fügt ein *File*- oder *Folder*-Objekt ein.

`FUNCTION` **`Remove`**`(name: STRING): …`
: entfernt ein *File*- oder *Folder*-Objekt aus dem Verzeichnis und liefert dieses Objekt als Rückgabewert.

`PROCEDURE` **`Delete`**`(name: STRING)`
: löscht ein *File*- oder *Folder*-Objekt aus dem Verzeichnis.

`FUNCTION` **`Size`**`: LONGINT`
: liefert die gesamte Größe des Verzeichnisses, d.h. die Summe der Größe aller darin enthaltenen Elemente.

Hinweis zu *Folder*: Die in einem Verzeichnis verwalteten Verzeichnisse und Dateien müssen nicht sortiert verwaltet werden. Darüber hinaus können Sie auch von einer fixen maximalen Anzahl von Elementen in einem Verzeichnis ausgehen, d.h. ein „einfaches“ Feld fixer Größe ist ausreichend.

Testen Sie Ihre Lösung ausführlich und beschreiben Sie kurz in welchem Bereich Ihrer Lösung Polymorphismus zum Einsatz kommt und welcher Vorteil daraus entsteht.

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.

Lösungsidee:

An sich hat sich hier Lösungstechnisch nichts geändert man muss einfach die Typen-Schnittstelle so umändern dass man Vektoren Objekte erstellen kann.

Testfälle:

```
Create new Vector v1 and fill it with numbers
1: 20
2: 3
3: 3
4: 3
5: 3
6: 3
7: 99
8: 0
9: 0
10: 0
Size of Vector: 7
Size After Clear: 0
Value at pos 4: 3
Write Vector
1: 0
2: 0
3: 0
4: 0
5: 0
6: 0
7: 0
8: 0
9: 0
10: 0
Create and Write another Vector with size 5
1: 0
2: 0
3: 0
4: 0
5: 0
```

Quellcode:

```
UNIT ModVectorObj;

INTERFACE

  TYPE
    data = ARRAY [1..1] OF INTEGER;
    VectorPtr = ^Vector;
    Vector = OBJECT
```

```
      PUBLIC
        CONSTRUCTOR Init(Capacity: INTEGER);
        PROCEDURE WriteV;
        PROCEDURE Add(val:INTEGER);
        FUNCTION Size: INTEGER;
        PROCEDURE InsertElementAt(pos: INTEGER; val: INTEGER; VAR ok: BOOLEAN);
        PROCEDURE GetElementAt(pos: INTEGER; VAR val: INTEGER; VAR ok: BOOLEAN);
        PROCEDURE Clear;
        FUNCTION Capacity: INTEGER;
      PRIVATE
        VCap: INTEGER;
        DataPtr: ^data;
    END;

IMPLEMENTATION

  CONSTRUCTOR Vector.Init(Capacity: INTEGER);
    VAR
      i: INTEGER;
  BEGIN
    GetMem(DataPtr,(Capacity+1) * SizeOf(INTEGER));
    SELF.VCap := Capacity;
    {$R-}
    FOR i := 1 TO SELF.VCap DO BEGIN
      SELF.DataPtr^[i] := 0;
    END; (* FOR *)
    {$R+}
  END;

  PROCEDURE Vector.WriteV;
    VAR
      i: INTEGER;
  BEGIN (* WriteV *)
    {$R-}
    FOR i := 1 TO SELF.VCap DO BEGIN
      WriteLn(i,': ',SELF.DataPtr^[i])
    END; (* FOR *)
    {$R+}
  END; (* WriteV *)

  PROCEDURE Vector.Add(val:INTEGER);
    VAR
     i: INTEGER;
  BEGIN
    i := 1;
```

```
    {$R-}
    WHILE (SELF.DataPtr^[i] <> 0) AND (i <= SELF.VCap) DO BEGIN
      Inc(i);
    END; (* WHILE *)
    {$R+}
    IF (i > SELF.VCap) THEN BEGIN
      WriteLn('No more Space!');
      HALT;
    END ELSE BEGIN
      {$R-}
      SELF.DataPtr^[i] := val;
      {$R+}
    END;
  END;

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

  PROCEDURE Vector.InsertElementAt(pos: INTEGER; val: INTEGER; VAR ok: BOOLEAN);
    VAR
      j,i, prev, next: INTEGER;
  BEGIN
    i := 1;
    j := pos;
    {$R-}
    WHILE (DataPtr^[i] = 0) AND (i <= VCap)DO BEGIN
      Inc(i);
    END; (* WHILE *)
    IF (i >= VCap) THEN BEGIN
      ok := FALSE;
    END ELSE IF (pos <= 0) THEN BEGIN
      j := 1;
      ok := TRUE;
```

```
      prev :=  DataPtr^[j];
      DataPtr^[j] := val;
      WHILE (DataPtr^[j+1] <> 0) DO BEGIN
        next := DataPtr^[j+1];
        DataPtr^[j+1] := prev;
        Inc(j);
        prev := next;
      END; (* WHILE *)
    END ELSE IF (pos > SELF.Size) THEN BEGIN
      ok := TRUE;
      Add(val);
    END ELSE BEGIN
      ok := TRUE;
      prev :=  DataPtr^[j];
      DataPtr^[j] := val;
      WHILE (DataPtr^[j+1] <> 0) DO BEGIN
        next := DataPtr^[j+1];
        DataPtr^[j+1] := prev;
        Inc(j);
        prev := next;
      END; (* WHILE *)
    END;
    {$R+}
  END;

  PROCEDURE Vector.GetElementAt(pos: INTEGER; VAR val: INTEGER; VAR ok: BOOLEAN);
  BEGIN
    {$R-}
    IF (pos > VCap) OR (pos < 1)THEN BEGIN
      ok := FALSE;
    END ELSE BEGIN
      ok := TRUE;
      val := DataPtr^[pos];
    END;
    {$R+}
  END;

  PROCEDURE Vector.Clear;
    VAR
      i,curSize: INTEGER;
  BEGIN
    curSize := SELF.Size;
    {$R-}
    FOR i := 1 TO curSize DO BEGIN
      DataPtr^[i] := 0;
```

```
    END; (* FOR *)
    {$R+}
  END;

  FUNCTION Vector.Capacity: INTEGER;
  BEGIN
    Capacity := VCap;
  END;

BEGIN (* ModVectorObj *)

END. (* ModVectorObj *)


PROGRAM Vtest;

USES ModVectorObj;

VAR
  v: VectorPtr;
  v2: VectorPtr;
  ok: BOOLEAN;
  test: INTEGER;
BEGIN (* Vtest *)
  WriteLn('Create new Vector v1 and fill it with numbers');
  New(v,Init(10));
  v^.Add(20);
  v^.Add(3);
  v^.Add(3);
  v^.Add(3);
  v^.Add(3);
  v^.Add(3);
  v^.InsertElementAt(10,99,ok);
  v^.WriteV;
  v^.GetElementAt(4,test,ok);
  WriteLn('Size of Vector: ',v^.Size);
  v^.Clear;
  WriteLn('Size After Clear: ',v^.Size);
  WriteLn('Value at pos 4: ',test);
  WriteLn('Write Vector');
  v^.WriteV;
  WriteLn('Create and Write another Vector with size 5');
  New(v2,Init(5));
  v2^.WriteV;
END. (* Vtest *)
```

Lösungsidee:

Hier muss man 3 Klassen erstellen. Stringbuilder ist hierbei die Basis Klasse von der TabStingbuider und Stringjoiner die meisten Methoden und auch den data String erben. Abgesehen davon ist das Implementieren der Methoden nicht wirklich schwer. Um Zeit und Code-Zeilen zu sparen kann man hier die geerbten Methoden benutzen und modifizieren(mittels Virtual) falls nötig. Beim TabStringbuilder muss man z.B jedes Mal bevor man eine geerbte Methode zum „Appenden“ von etwas aufruft, dass die gewünschte Anzahl an Leerzeichen eingefügt werden. Ähnlich ist es beim StringJoiner wobei hier man den delimeter anstatt der Leerzeichen einsetzten muss.

Testfälle:

```
Testing StringBuilder
eins 2 TRUE

Testing TabStringBuilder
Eins        2        TRUE

Testing StringJoiner
Eins,Zwei,Drei
```

Quellcode:

```
UNIT ModStringBuilder;

INTERFACE
  TYPE
    StringBuilderPtr = ^StringBuilder;
    StringBuilder = OBJECT
      PUBLIC
        CONSTRUCTOR Init;
        FUNCTION CheckIfStart: BOOLEAN;
        PROCEDURE AppendStr(e:STRING); VIRTUAL;
        PROCEDURE AppendChar(e: CHAR); VIRTUAL;
        PROCEDURE AppendInt(e:INTEGER); VIRTUAL;
        PROCEDURE AppendBool(e:BOOLEAN); VIRTUAL;
        FUNCTION AsString: STRING;
      PRIVATE
        data: STRING;
    END;
```

```
    TabStringBuilderPtr = ^TabStringBuilder;
    TabStringBuilder = OBJECT(StringBuilder)
      PUBLIC
        CONSTRUCTOR Init(count: INTEGER);
        PROCEDURE AppendStr(e:STRING); VIRTUAL;
        PROCEDURE AppendChar(e: CHAR); VIRTUAL;
        PROCEDURE AppendInt(e:INTEGER); VIRTUAL;
        PROCEDURE AppendBool(e:BOOLEAN); VIRTUAL;
      PRIVATE
        cnt: INTEGER;
    END;

    StringJoinerPtr = ^StringJoiner;
    StringJoiner = OBJECT(StringBuilder)
      PUBLIC
        CONSTRUCTOR Init(d: CHAR);
        PROCEDURE Add(e:STRING);
      PRIVATE
        delim: STRING;
    END;

IMPLEMENTATION

  CONSTRUCTOR StringBuilder.Init;
  BEGIN
    data := '';
  END;

  FUNCTION StringBuilder.CheckIfStart: BOOLEAN;
  BEGIN (* StringBuilder.CheckIfStart *)
    IF (data = '') THEN BEGIN
      CheckIfStart := TRUE
    END ELSE BEGIN
      CheckIfStart := FALSE;
    END;
  END; (* StringBuilder.CheckIfStart *)

  PROCEDURE StringBuilder.AppendStr(e:STRING);
  BEGIN (* StringBuilder.AppendStr *)
    data := data + e;
  END; (* StringBuilder.AppendStr *)

  PROCEDURE StringBuilder.AppendChar(e:CHAR);
  BEGIN (* StringBuilder.AppendChar *)
    data := data + e;
```

```
END; (* StringBuilder.AppendChar *)

PROCEDURE StringBuilder.AppendInt(e: INTEGER);
  VAR
    s: STRING;
BEGIN (* StringBuilder.AppendInt *)
  Str(e,s);
  data := data + s;
END; (* StringBuilder.AppendInt *)

PROCEDURE StringBuilder.AppendBool(e:BOOLEAN);
BEGIN (* StringBuilder.AppendBool *)
  IF (e = TRUE) THEN BEGIN
    data := data + 'TRUE';
  END ELSE BEGIN
    data := data + 'FALSE';
  END;
END; (* StringBuilder.AppendBool *)

FUNCTION StringBuilder.AsString: STRING;
BEGIN (* AsString *)
  AsString := data;
END; (* AsString *)

{-----------------TabStringBuilder-----------------}

CONSTRUCTOR TabStringBuilder.Init(count: INTEGER);
BEGIN
  INHERITED Init;
  SELF.cnt := count;
END;

PROCEDURE TabStringBuilder.AppendStr(e: STRING);
  VAR
    i: INTEGER;
BEGIN (* TabStringBuilder.AppendSt *)
  IF (INHERITED CheckIfStart) THEN BEGIN
    INHERITED AppendStr(e);
  END ELSE BEGIN
    FOR i := 1 TO cnt DO BEGIN
      data := data + ' ';
    END; (* FOR *)
    INHERITED AppendStr(e);
  END;
END; (* TabStringBuilder.AppendStr *)
```

```
PROCEDURE TabStringBuilder.AppendChar(e: CHAR);
  VAR
    i: INTEGER;
BEGIN (* TabStringBuilder.AppendChar *)
  IF (INHERITED CheckIfStart) THEN BEGIN
    INHERITED AppendChar(e);
  END ELSE BEGIN
    FOR i := 1 TO cnt DO BEGIN
      data := data + ' ';
    END; (* FOR *)
    INHERITED AppendChar(e);
  END;
END; (* TabStringBuilder.AppendChar *)

PROCEDURE TabStringBuilder.AppendInt(e: INTEGER);
  VAR
    i: INTEGER;
    s: STRING;
BEGIN (* TabStringBuilder.AppendInt *)
  IF (INHERITED CheckIfStart) THEN BEGIN
    INHERITED AppendInt(e);
  END ELSE BEGIN
    FOR i := 1 TO cnt DO BEGIN
      data := data + ' ';
    END; (* FOR *)
    INHERITED AppendInt(e);
  END;
END; (* TabStringBuilder.AppendInt *)

PROCEDURE TabStringBuilder.AppendBool(e: BOOLEAN);
  VAR
    i: INTEGER;
BEGIN (* TabStringBuilder.AppendBool *)
  IF (INHERITED CheckIfStart) THEN BEGIN
    INHERITED AppendBool(e);
  END ELSE BEGIN
    FOR i := 1 TO cnt DO BEGIN
      data := data + ' ';
    END; (* FOR *)
    INHERITED AppendBool(e);
  END;
END; (* TabStringBuilder.AppendBool *)

{-----------------StringJoiner-----------------}
```

```
  CONSTRUCTOR StringJoiner.Init(d:CHAR);
  BEGIN
    INHERITED Init;
    delim := d;
  END;

  PROCEDURE StringJoiner.Add(e:STRING);
  BEGIN (* StringJoiner.Add *)
    IF (INHERITED CheckIfStart) THEN BEGIN
      INHERITED AppendStr(e);
    END ELSE BEGIN
      data := data + delim;
      INHERITED AppendStr(e);
    END;
  END; (* StringJoiner.Add *)

BEGIN (* ModStringBuilder *)

END. (* ModStringBuilder *)


PROGRAM StringTest;

USES ModStringBuilder;


VAR
  s: StringBuilderPtr;
  t: TabStringBuilderPtr;
  j: StringJoinerPtr;
BEGIN (* StringTest *)
  WriteLn('Testing StringBuilder');
  New(s,Init);
  s^.AppendStr('eins');
  s^.AppendChar(' ');
  s^.AppendInt(2);
  s^.AppendChar(' ');
  s^.AppendBool(TRUE);
  WriteLn(s^.AsString);
  WriteLn;

  WriteLn('Testing TabStringBuilder');
  New(t, Init(8));
  t^.AppendStr('Eins');
  t^.AppendInt(2);
```

```
  t^.AppendBool(TRUE);
  WriteLn(t^.AsString);
  WriteLn;

  WriteLn('Testing StringJoiner');
  New(j, Init(','));
  j^.Add('Eins');
  j^.Add('Zwei');
  j^.Add('Drei');
  WriteLn(j^.AsString);
END. (* StringTest *)
```

Lösungsidee:

Man soll ein Dateisystem als Klassen implementieren. Hier kann man ganz gut Polymorphie anwenden wenn man z.B in einem Folder etwas speichert. Beide Klassen Files und Folders erben von der Klasse data. Im folder befindet sich ein Array, dass Objekte der Klasse Data speichern. Da Files und Folder auch Data sind ist es dem Array egal ob da jetzt Files oder Folder gespeichert werden. Letzten Endes können sich Folder und File Objekte auch als Data Objekte verhalten. Das ist der große Vorteil von Polymorphie.

Testfälle:

```
Create Fi1.png File
Create Fi2.png File
Create Fo1 Folder
Create Fo2 Folder
Add fi1 into fo1
Add fo1 into fo2
Add fi2 into fo2

Write Data of fo1
1: fi1

Remove f1 of fo1 and return it
Name of returned Data: fi1
Check if f1 really got removed from fo1 by Writing its Data
Write Data of fo2
1: fo1
2: fi1
Delete Data in fo2 called fo1
Check if fo1 got deleted by Writing its data
Write fo2 Data
1: fi1
```

Quellcode:

```
UNIT ModData;

INTERFACE

  USES sysutils;

  TYPE
    DataPtr = ^Data;
    Data = OBJECT
    PUBLIC
      CONSTRUCTOR Init(name:STRING;Typestr:STRING);
      DESTRUCTOR Done;VIRTUAL;
      PROCEDURE UpdateDate;
      FUNCTION ReturnName: STRING;
    PRIVATE
      name: STRING;
      Typestr: STRING;
      dateModified: STRING;
  END;

IMPLEMENTATION

  CONSTRUCTOR Data.Init(name,Typestr: STRING);
  BEGIN
    SELF.name := name;
    SELF.Typestr := Typestr;
    SELF.dateModified := DateTimeToStr(Now);
  END;

  DESTRUCTOR Data.Done;
  BEGIN

  END;

  PROCEDURE Data.UpdateDate;
  BEGIN (* Data.UpdateDate *)
    SELF.dateModified := DateTimeToStr(Now);
  END; (* Data.UpdateDate *)

  FUNCTION Data.ReturnName: STRING;
  BEGIN (* ReturnName *)
    ReturnName := name;
  END; (* ReturnName *)
```

```
BEGIN (* ModData *)

END. (* ModData *)


UNIT ModFile;

INTERFACE

  USES ModData;

  TYPE
    FilesPtr = ^Files;
    Files = OBJECT(data)
      PUBLIC
        CONSTRUCTOR Init(name,typestr: STRING);
      PRIVATE
        size: LONGINT;
    END;

IMPLEMENTATION

  CONSTRUCTOR Files.Init(name,typestr: STRING);
  BEGIN
    INHERITED Init(name,typestr);
  END;

BEGIN (* ModFile *)

END. (* ModFile *)


UNIT ModFolder;

INTERFACE

  USES ModData,ModFile,sysutils;

  CONST
    maxSize = 30;

  TYPE
    FolderPtr = ^Folder;
    Folder = OBJECT(Data)
    PUBLIC
```

```
      CONSTRUCTOR Init(name:STRING;Typestr:STRING);
      DESTRUCTOR Done;VIRTUAL;
      PROCEDURE Add(fileObj: DataPtr);
      FUNCTION Remove(name: STRING):DataPtr;
      PROCEDURE Delete(name: STRING);
      PROCEDURE WriteData;
      FUNCTION Size: LONGINT;
    PRIVATE
      data : ARRAY[1..maxSize] OF DataPtr;
      top: INTEGER;
  END;

IMPLEMENTATION

  CONSTRUCTOR Folder.Init(name,Typestr: STRING);
  BEGIN
    INHERITED Init(name,Typestr);
    top := 0;
  END;

  DESTRUCTOR Folder.Done;
    VAR
      i: INTEGER;
  BEGIN
    INHERITED Done;
    FOR i := 1 TO top DO BEGIN
      Dispose(data[i],Done);
    END; (* FOR *)
  END;

  PROCEDURE Folder.Add(FileObj: DataPtr);
  BEGIN (* add *)
    Inc(top);
    IF (top > maxSize) THEN BEGIN
      WriteLn('Data is Full');
      HALT;
    END; (* IF *)
    data[top] := FileObj;
    UpdateDate;
  END; (* add *)

  FUNCTION Folder.Remove(name: STRING):DataPtr;
    VAR
      i,j: INTEGER;
      temp: DataPtr;
```

```
BEGIN (* Folder.Remove *)
  i := 1;
  WHILE (i <= top) AND (data[i]^.ReturnName <> name) DO BEGIN
    Inc(i);
  END;
  IF (data[i]^.ReturnName = name) THEN BEGIN
    temp := data[i];
    FOR j := i TO top DO BEGIN
      data[j] := data[j+1];
    END; (* FOR *)
    Dec(top);
    Remove := temp;
  END ELSE BEGIN
    WriteLn('Data doesnt exist');
    HALT;
  END;
  UpdateDate;
END; (* Folder.Remove *)

PROCEDURE Folder.Delete(name: STRING);
  VAR
    i,j: INTEGER;
    temp: DataPtr;
BEGIN (* Folder.Remove *)
  i := 1;
  WHILE (i <= top) AND (data[i]^.ReturnName <> name) DO BEGIN
    Inc(i);
  END;
  IF (data[i]^.ReturnName = name) THEN BEGIN
    Dispose(data[i],Done);
    FOR j := i TO top DO BEGIN
      data[j] := data[j+1];
    END; (* FOR *)
    Dec(top);
  END ELSE BEGIN
    WriteLn('Data doesnt exist');
    HALT;
  END;
  UpdateDate;
END; (* Folder.Remove *)

FUNCTION Folder.Size: LONGINT;
BEGIN (* Folder.Size *)
  Size := top;
END; (* Folder.Size *)
```

```
  PROCEDURE Folder.WriteData;
    VAR
      i: INTEGER;
  BEGIN (* WriteData *)
    FOR i := 1 TO top DO BEGIN
      WriteLn(i,': ',data[i]^.ReturnName);
    END; (* FOR *)
  END; (* WriteData *)


BEGIN (* ModData *)

END. (* ModData *)
```