Abgabetermin: Mi, 31. 05. 2023

☒ **Gr. 1,** Dr. S. Wagner **Name** Elias Wurm **Aufwand in h** 4
☐ **Gr. 2,** Dr. D. Auer
☐ **Gr. 3,** Dr. G. Kronberger **Punkte** _____ **Kurzzeichen Tutor / Übungsleiter*in** _____ / _____

1. Verkettete Listen (5 + 3 Punkte)

Nachdem Sie im ersten Semester verkettete Listen imperativ implementiert haben, ist nun eine objektorientierte Implementierung von einfach-verketteten Listen ohne Ankerelement gesucht, die ganze Zahlen aufnehmen können.

a) Implementieren Sie eine Klasse `List` mit zumindest folgenden Methoden:

`PROCEDURE Add(val: INTEGER);`
fügt `val` hinten in die Liste ein.

`FUNCTION Contains(val: INTEGER): BOOLEAN;`
ermittelt, ob `val` in der Liste enthalten ist.

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

`PROCEDURE Remove(val: INTEGER);`
löscht alle Elemente mit dem Wert `val` aus der Liste.

`PROCEDURE Clear;`
leert den Behälter (`Size` ist dann 0).

b) Leiten Sie nun von `List` eine neue Klasse `SortedList` ab. Darin

- überschreiben Sie die Methode `Add` so, dass die Elemente in aufsteigend sortierter Reihenfolge eingefügt werden und
- überschreiben Sie die Methode `Contains` so, dass diese von der Sortierung Gebrauch macht. Wie wirkt sich diese Änderung auf die asymptotische Laufzeitkomplexität dieser Methode aus?

2. Klassen für Zeichenketten-Operationen (3 + 3 + 2 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 (8 Punkte)

Entwickeln Sie die erforderlichen Klassen, um ein Dateisystem (`File` und `Folder`) zu beschreiben und zu simulieren (es sollen keine echten Dateien erzeugt, verschoben oder gelöscht werden!). 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.

Aufgabe 1 – Verkettete Liste

Lösungsidee:

Die Liste verwendet eine einfach verkettete Liste, in der die Elemente in der Reihenfolge ihres Einfügens hinzugefügt werden.

Die sortierte Liste verwendet eine einfach verkettete Liste, in der die Elemente in aufsteigend sortierter Reihenfolge eingefügt werden. Beim Hinzufügen eines neuen Elements wird es an der richtigen Position eingefügt, um die Sortierung beizubehalten.

Laufzeitkomplexität:

In der List wird eine einfache lineare Suche durchgeführt, indem jedes Element der Liste nacheinander überprüft wird. Die Laufzeitkomplexität ist daher linear (O(n)), da alle Elemente durchlaufen werden müssen, um das gesuchte Element zu finden oder festzustellen, dass es nicht vorhanden ist.

In der SortedList wird hingegen eine optimierte Suche verwendet, um das Element effizient zu finden. Der Code nutzt die Sortierung der Liste aus, um die Suche vorzeitig abzubrechen, wenn der aktuelle Wert größer als value ist. Dadurch wird die Anzahl der durchlaufenden Elemente reduziert und die Laufzeitkomplexität auf O(log n) reduziert.

Zeitaufwand: ~1h

Code:

```
unit ListUnit;

interface

type
  ListPtr = ^ListNode;
  ListNode = object
    value: Integer;
    next: ListPtr;
  end;

  List = ^ListObj;
  ListObj = object
  public
    constructor Init;
    destructor Done; virtual;

    procedure Add(val: Integer);
    function Contains(val: Integer): Boolean;
```

```
    function Size: Integer;
    procedure Remove(val: Integer);
    procedure Clear;
  protected
    head: ListPtr;
    function NewNode(val: integer): ListPtr;
  end;

function NewList: List;

implementation

function NewList: List;
var
  l: List;
begin
  New(l, Init);
  NewList := l;
end;

constructor ListObj.Init;
begin
  head := nil;
end;

destructor ListObj.Done;
begin
  Clear;
end;

procedure ListObj.Add(val: Integer);
var
  node, lastNode: ListPtr;
begin
  node := NewNode(val);

  if head = nil then
    head := node
  else
  begin
    lastNode := head;
    while lastNode^.next <> nil do
      lastNode := lastNode^.next;
    lastNode^.next := node;
  end;
end;

function ListObj.Contains(val: Integer): Boolean;
```

```
var
  currentNode: ListPtr;
begin
  currentNode := head;
  while currentNode <> nil do
  begin
    if currentNode^.value = val then
    begin
      Contains := true;
      Exit;
    end;
    currentNode := currentNode^.next;
  end;
  Contains := false;
end;

function ListObj.Size: Integer;
var
  currentNode: ListPtr;
  i: integer;
begin
  currentNode := head;
  i := 0;
  while currentNode <> nil do
  begin
    Inc(i);
    currentNode := currentNode^.next;
  end;
  Size := i;
end;

procedure ListObj.Remove(val: Integer);
var
  currentNode, prevNode, tempNode: ListPtr;
begin
  prevNode := nil;
  currentNode := head;

  while currentNode <> nil do
    if currentNode^.value = val then
    begin
      if prevNode = nil then
      begin
        // The node to remove is the head node
        tempNode := head;
        head := currentNode^.next;
        Dispose(tempNode);
        currentNode := head;
```

```
      end
      else
      begin
        // The node to remove is not the head node
        tempNode := currentNode;
        prevNode^.next := currentNode^.next;
        currentNode := currentNode^.next;
        Dispose(tempNode);
      end;
    end
    else
    begin
      prevNode := currentNode;
      currentNode := currentNode^.next;
    end;
end;

procedure ListObj.Clear;
var
  currentNode, tempNode: ListPtr;
begin
  currentNode := head;
  while currentNode <> nil do
  begin
    tempNode := currentNode;
    currentNode := currentNode^.next;
    Dispose(tempNode);
  end;
  head := nil;
end;

function ListObj.NewNode(val: integer): ListPtr;
var
  node: ListPtr;
begin
  New(node);
  node^.value := val;
  node^.next := nil;
  NewNode := node;
end;

end.
```

```
unit SortedListUnit;

interface
```

```
uses
  ListUnit;

type
  SortedList = ^SortedListObj;
  SortedListObj = object(ListObj)
    constructor Init;
    destructor Done; virtual;

    procedure Add(val: Integer); virtual;
    function Contains(val: Integer): Boolean; virtual;
  end;

function NewSortedList: SortedList;

implementation

function NewSortedList: SortedList;
var
  sl: SortedList;
begin
  New(sl, Init);
  NewSortedList := sl;
end;

constructor SortedListObj.Init;
begin
  inherited Init;
end;

destructor SortedListObj.Done;
begin
  inherited Done;
end;

procedure SortedListObj.Add(val: Integer);
var
  node, currentNode, prevNode: ListPtr;
begin
  node := NewNode(val);

  if head = nil then
    head := node
  else if val < head^.value then
  begin
    node^.next := head;
    head := node;
  end
```

```
  else
  begin
    prevNode := head;
    currentNode := head^.next;

    while (currentNode <> nil) and (currentNode^.value < val) do
    begin
      prevNode := currentNode;
      currentNode := currentNode^.next;
    end;

    node^.next := currentNode;
    prevNode^.next := node;
  end;
end;

function SortedListObj.Contains(val: Integer): Boolean;
var
  currentNode: ListPtr;
begin
  currentNode := head;

  while (currentNode <> nil) and (currentNode^.value < val) do
    currentNode := currentNode^.next;

  Contains := (currentNode <> nil) and (currentNode^.value = val);
end;

end.
```

Test:

```
program TestList;

uses
  ListUnit, SortedListUnit;

procedure ExecuteListTests(l: List);
begin
  // Add some values to the list
  l^.Add(5);
  l^.Add(10);
  l^.Add(15);

  // Print the size of the list
  Writeln('Size of the list: ', l^.Size);

  // Check if the list contains a value
```

```
  Writeln('List contains 10: ', l^.Contains(10));
  Writeln('List contains 20: ', l^.Contains(20));

  // Remove a value from the list
  l^.Remove(10);

  // Print the updated size of the list
  Writeln('Size of the list after removal: ', l^.Size);

  // Clear the list
  l^.Clear;
  Writeln('Size of the list after clearing: ', l^.Size);
end;

var
  l: List;
  sl: SortedList;

begin
  l := NewList;
  sl := NewSortedList;

  Writeln('Testing List:');
  ExecuteListTests(l);
  Dispose(l, Done);

  writeln;writeln;
  Writeln('Testing SortedList:');
  ExecuteListTests(sl);
  Dispose(sl, Done);
  writeln;writeln;
end.
```

```
 *  Executing task: C:\_data\fh-repos\2023SS_ADF\UE8\hu\TestList.exe

Testing List:
Size of the list: 3
List contains 10: TRUE
List contains 20: FALSE
Size of the list after removal: 2
Size of the list after clearing: 0


Testing SortedList:
Size of the list: 3
List contains 10: TRUE
List contains 20: FALSE
Size of the list after removal: 2
Size of the list after clearing: 0


Heap dump by heaptrc unit of C:\_data\fh-repos\2023SS_ADF\UE8\hu\TestList.exe
8 memory blocks allocated : 64/64
8 memory blocks freed     : 64/64
0 unfreed memory blocks : 0
True heap size : 98304 (112 used in System startup)
True free heap : 98192
```

Aufgabe 2 – Klassen für Zeichenketten-Operationen

Lösungsidee:

(a) Die Klasse StringBuilder enthält eine Datenkomponente buffer vom Typ STRING sowie die weiteren Methoden zum Anhängen verschiedener Typen an den String.

(b) Die abgeleitete Klasse TabStringBuilder erbt von der Klasse StringBuilder und überschreibt die Methoden, um Elemente spaltenweise mit Leerzeichen aufzurichten, indem Sie jedes Mal den hereinkommenden Typen so bearbeitet das er genau 1 spalte breit ist und den Rest mit Leerzeichen befüllt. Die Spaltenbreite wird beim Erstellen eines TabStringBuilder-Objekts festgelegt.

(c) Die Klasse StringJoiner verwendet eine Datenkomponente vom Typen StringBuilder zur Verkettung von Zeichenketten und Trennzeichen. Sie enthält einen Konstruktor Init, um ein StringJoiner-Objekt mit einem Trennzeichen delimiter zu initialisieren, die Methode Add, um eine Zeichenkette e mit dem Trennzeichen an den StringJoiner anzufügen, und die Methode AsString, um das Ergebnis als Zeichenkette zurückzugeben.

Zeitaufwand: ~1h

Code:

```
unit StringBuilderUnit;

interface

type
  StringBuilderPtr = ^StringBuilderObj;
  StringBuilderObj = object
  public
    constructor Init;
    destructor Done; virtual;

    procedure AppendStr(e: string); virtual;
    procedure AppendChar(e: char); virtual;
    procedure AppendInt(e: integer); virtual;
    procedure AppendBool(e: boolean); virtual;
    function AsString: string; virtual;
    function BufferLength: integer;
  private
```

```
    buffer: string;
  end;

function NewStringBuilder: StringBuilderPtr;

implementation

function NewStringBuilder: StringBuilderPtr;
var
  builder: StringBuilderPtr;
begin
  New(builder, init);
  NewStringBuilder := builder;
end;

constructor StringBuilderObj.Init;
begin
  buffer := '';
end;

destructor StringBuilderObj.Done;
begin
end;

procedure StringBuilderObj.AppendStr(e: string);
begin
  buffer := buffer + e;
end;

procedure StringBuilderObj.AppendChar(e: char);
begin
  buffer := buffer + e;
end;

procedure StringBuilderObj.AppendInt(e: integer);
```

```
var
  intStr: string;
begin
  Str(e, intStr);
  buffer := buffer + intStr;
end;

procedure StringBuilderObj.AppendBool(e: boolean);
begin
  if e then
    buffer := buffer + 'TRUE'
  else
    buffer := buffer + 'FALSE';
end;

function StringBuilderObj.AsString: string;
begin
  AsString := buffer;
end;

function StringBuilderObj.BufferLength: integer;
begin
  BufferLength := Length(buffer);
end;

end.
```

```
unit TabStringBuilderUnit;

interface

uses
  StringBuilderUnit;
```

```
type
  TabStringBuilderPtr = ^TabStringBuilderObj;
  TabStringBuilderObj = object(StringBuilderObj)
  public
    constructor Init(width: integer);
    destructor Done; virtual;
    procedure AppendStr(e: string); virtual;
    procedure AppendChar(e: char); virtual;
    procedure AppendInt(e: integer); virtual;
    procedure AppendBool(e: boolean); virtual;
  private
    columnWidth: integer;
    function AlignText(text: string): string;
  end;

function NewTabStringBuilder(width: integer):
TabStringBuilderPtr;

implementation

function NewTabStringBuilder(width: integer):
TabStringBuilderPtr;
var
  builder: TabStringBuilderPtr;
begin
  New(builder, Init(width));
  NewTabStringBuilder := builder;
end;

constructor TabStringBuilderObj.Init(width: integer);
begin
  inherited Init;
  columnWidth := width;
end;
```

```
destructor TabStringBuilderObj.Done;
begin
  inherited done;
end;

procedure TabStringBuilderObj.AppendStr(e: string);
begin
  inherited AppendStr(AlignText(e));
end;

procedure TabStringBuilderObj.AppendChar(e: char);
begin
  inherited AppendStr(AlignText(e));
end;

procedure TabStringBuilderObj.AppendInt(e: integer);
var
  intStr: string;
begin
  Str(e, intStr);
  inherited AppendStr(AlignText(intStr));
end;

procedure TabStringBuilderObj.AppendBool(e: boolean);
begin
  if e then
    inherited AppendStr(AlignText('TRUE'))
  else
    inherited AppendStr(AlignText('FALSE'));
end;

function TabStringBuilderObj.AlignText(text: string):
string;
var
  temp: string;
```

```
begin
  if Length(text) >= columnWidth then
    temp := Copy(text, 1, columnWidth)
  else
  begin
    temp := text;
    while Length(temp) < columnWidth do
      temp := Concat(temp, ' ');
  end;
  AlignText := temp;
end;

end.
```

```
unit StringJoinerUnit;

interface

uses
  StringBuilderUnit;

type
  StringJoinerPtr = ^StringJoinerObj;
  StringJoinerObj = object
  public
    constructor Init(delimiter: char);
    destructor Done; virtual;

    procedure Add(e: string);
    function AsString: string;
  private
    delimiter: char;
    count: integer;
    resultBuilder: StringBuilderPtr;
```

```
  end;

function NewStringJoiner(delimiter: char):
StringJoinerPtr;

implementation

function NewStringJoiner(delimiter: char):
StringJoinerPtr;
var
  joiner: StringJoinerPtr;
begin
  New(joiner, Init(delimiter));
  NewStringJoiner := joiner;
end;

constructor StringJoinerObj.Init(delimiter: char);
begin
  self.delimiter := delimiter;
  count := 0;
  resultBuilder := NewStringBuilder;
end;

destructor StringJoinerObj.Done;
begin
  Dispose(resultBuilder, Done);
end;

procedure StringJoinerObj.Add(e: string);
begin
  if count > 0 then
    resultBuilder^.AppendChar(delimiter);
  resultBuilder^.AppendStr(e);
  Inc(count);
end;
```

```
function StringJoinerObj.AsString: string;
begin
  AsString := resultBuilder^.AsString;
end;

end.
```

Test:

```
program TestStringBuilder;

uses
  StringBuilderUnit, TabStringBuilderUnit;

procedure ExecuteStringBuilderTests(builder:
StringBuilderPtr);
begin
  // Append different types of values to the
StringBuilder
  builder^.AppendStr('Hello ');
  builder^.AppendChar('W');
  builder^.AppendChar('o');
  builder^.AppendChar('r');
  builder^.AppendChar('l');
  builder^.AppendChar('d');
  builder^.AppendInt(2023);
  builder^.AppendBool(true);
  builder^.AppendStr('123456789');

  // Get the resulting string from the StringBuilder
  Writeln('StringBuilder content: ', builder^.AsString);
end;
```

```
var
  myBuilder: StringBuilderPtr;
  myTabBuilder: TabStringBuilderPtr;

begin
  myBuilder := NewStringBuilder;
  myTabBuilder := NewTabStringBuilder(8);

  Writeln('Testing StringBuilder:');
  ExecuteStringBuilderTests(myBuilder);

  Dispose(myBuilder, Done);

  writeln; writeln;
  Writeln('Testing TabStringBuilder:');
  ExecuteStringBuilderTests(myTabBuilder);

  Dispose(myTabBuilder, Done);
  writeln; writeln;
end.
```

```
Testing StringBuilder:
StringBuilder content: Hello World2023TRUE123456789


Testing TabStringBuilder:
StringBuilder content: Hello   W       o       r       l       d       2023    TRUE    12345678


Heap dump by heaptrc unit of C:\_data\fh-repos\2023SS_ADF\UE8\hu2\TestStringBuilder.exe
2 memory blocks allocated : 524/528
2 memory blocks freed     : 524/528
0 unfreed memory blocks : 0
True heap size : 98304 (112 used in System startup)
True free heap : 98192
```

```
program TestStringJoiner;

uses
  StringJoinerUnit;

procedure ExeuteStringJoinerTests;
```

```
var
  joiner: StringJoinerPtr;
begin
  // Create a StringJoiner with delimiter ","
  joiner := NewStringJoiner(',');

  // Add some strings
  joiner^.Add('Hello');
  joiner^.Add('World');
  joiner^.Add('!');
  joiner^.Add('How');
  joiner^.Add('are');
  joiner^.Add('');
  joiner^.Add('you');
  joiner^.Add('today');
  joiner^.Add('?');

  // Get and print the result
  Writeln('Result: ', joiner^.AsString);

  // Clean up memory
  Dispose(joiner, done);
end;

begin
  ExeuteStringJoinerTests;
end.
```

```
Result: Hello,World,!,How,are,,you,today,?
Heap dump by heaptrc unit of C:\_data\fh-repos\2023SS_ADF\UE8\hu2\TestStringJoiner.exe
2 memory blocks allocated : 272/280
2 memory blocks freed     : 272/280
0 unfreed memory blocks : 0
True heap size : 131072 (112 used in System startup)
True free heap : 130960
```

Aufgabe 3 – Dateisystem als Klassen

Lösungsidee:

Ich verwende für diese Aufgabe das Composite Pattern, indem ich eine Basis Klasse namens Entity implementiere und jeweils Folder als auch File erben von dieser Basis Klasse, sodass Folder eine Sammlung an Entities hat und Folder und Files gleich behandelt werden können für Operationen wie Move, Delete, Add, ….

Zeitaufwand: ~2h

Code:

```
unit EntityUnit;

interface

uses sysUtils;

type
  EntityType = (FileType, FolderType);

  EntityPtr = ^EntityObj;
  EntityObj = object
  public
    constructor Init(name: string; entityType: EntityType);
    destructor Done; virtual;

    function AsString: string; virtual;

  public
    name: string;
    entityType: EntityType;
    dateModified: TDateTime;
  end;

implementation

constructor EntityObj.Init(name: string; entityType: EntityType);
begin
  self.name := name;
  self.entityType := entityType;
  dateModified := Now;
end;

destructor EntityObj.Done;
begin
end;
```

```
function EntityObj.AsString: string;
var
  typeStr: string;
begin
  case entityType of
    FileType: typeStr := 'file';
    FolderType: typeStr := 'folder';
  else
    typeStr := 'undefiend';
  end;

  AsString := 'name: ' + name + ', type: ' + typeStr + ', dateModified: ' +
DateTimeToStr(dateModified);
end;

end.
```

```
unit FileUnit;

interface

uses EntityUnit;

type
  FilePtr = ^FileObj;
  FileObj = object(EntityObj)
    size: longint;

    constructor Init(name: string; size: longInt);
    destructor Done; virtual;

    function AsString: string; virtual;
  end;

function NewFile(name: string; size: longInt): FilePtr;

implementation

function NewFile(name: string; size: longInt): FilePtr;
var
  f: FilePtr;
begin
  New(f, Init(name, size));
  NewFile := f;
end;
```

```
constructor FileObj.Init(name: string; size: longInt);
begin
  self.size := size;
  inherited Init(name, FileType);
end;

destructor FileObj.Done;
begin
  inherited Done;
end;

function FileObj.AsString: string;
var
  sizeStr: string;
begin
  Str(size, sizeStr);
  AsString := inherited + ' ,size: ' + sizeStr;
end;

end.
```

```
unit FolderUnit;

interface

uses EntityUnit, FileUnit, StringBuilderUnit;

const
  MAX_FOLDER_SIZE = 50;

type
  FolderPtr = ^FolderObj;
  FolderObj = object(EntityObj)
  public
    constructor Init(name: string);
    destructor Done; virtual;

    procedure Add(entity: EntityPtr);
    function Remove(name: STRING): EntityPtr;
    procedure Delete(name: STRING);
    procedure Move(name: string; destination: FolderPtr);
    function Size: longInt;

    function AsString: string; virtual;
  private
    children: array[0..MAX_FOLDER_SIZE] of EntityPtr;
    count: integer;
```

```
    function FindEmptySlot: integer;
    function FindIndexByName(name: string): integer;
  end;

function NewFolder(name: string): FolderPtr;

implementation

function NewFolder(name: string): FolderPtr;
var
  f: FolderPtr;
begin
  New(f, Init(name));
  NewFolder := f;
end;

constructor FolderObj.Init(name: string);
begin
  count := 0;
  inherited Init(name, FolderType);
end;

destructor FolderObj.Done;
var
  i: integer;
begin
  inherited Done;
  for i := Low(children) to High(children) do
    if (children[i] <> nil) then
      Dispose(children[i], Done);
end;

procedure FolderObj.Add(entity: EntityPtr);
begin
  if(count = MAX_FOLDER_SIZE) then
  begin
    writeln('ERROR: Max. folder size reached!');
    Halt;
  end;

  children[FindEmptySlot] := entity;
  Inc(count);
end;

function FolderObj.Remove(name: STRING): EntityPtr;
var
  i: Integer;
```

```
begin
  i := FindIndexByName(name);
  if i >= 0 then
  begin
    Remove := children[i];
    children[i] := nil;
    Dec(count);
  end else
    Remove := nil;
end;

procedure FolderObj.Delete(name: STRING);
var
  i: Integer;
begin
  i := FindIndexByName(name);
  if i >= 0 then
  begin
    Dispose(children[i], Done);
    children[i] := nil;
    Dec(count);
  end;
end;

procedure FolderObj.Move(name: string; destination: FolderPtr);
var
  entity: EntityPtr;
begin
  entity := Remove(name);
  if entity <> nil then
    destination^.Add(entity);
end;

function FolderObj.Size: longInt;
var
  i, sum: longInt;
begin
  sum := 0;
  for i := Low(children) to High(children) do
    if (children[i] <> nil) then
      case children[i]^.entityType of
        FileType: sum := sum + FilePtr(children[i])^.size;
        FolderType: sum := sum + FolderPtr(children[i])^.Size;
      end;
  Size := sum;
end;

function FolderObj.FindEmptySlot: Integer;
```

```
var
  i: Integer;
begin
  for i := Low(children) to High(children) do
    if children[i] = nil then
    begin
      FindEmptySlot := i;
      Exit;
    end;
end;

function FolderObj.FindIndexByName(name: string): integer;
var
  i: Integer;
begin
  if(count = 0) then
  begin FindIndexByName := -1; Exit; end;

  for i := Low(children) to High(children) do
    if (children[i] <> nil) and (children[i]^.name = name) then
    begin
      FindIndexByName := i;
      Exit;
    end;
  FindIndexByName := -1;
end;

function FolderObj.AsString: string;
var
  i: integer;
  strBuilder: StringBuilderPtr;
begin
  strBuilder := NewStringBuilder;

  strBuilder^.AppendStr(inherited AsString);
  strBuilder^.AppendStr(', childrenAmount:');
  strBuilder^.AppendInt(count);
  strBuilder^.AppendStr(', size:');
  strBuilder^.AppendLongInt(Size);
  strBuilder^.AppendStr(', children:');

  for i := Low(children) to High(children) do
    if children[i] <> nil then
    begin
      strBuilder^.AppendLine;
      strBuilder^.AppendStr('  ');
      strBuilder^.AppendStr(children[i]^.asString);
    end;
```

```
  AsString := strBuilder^.AsString;
  Dispose(strBuilder, Done);
end;

end.
```

Test:

```
program TestFS;

uses
  EntityUnit,
  FileUnit,
  FolderUnit;

var
  file1, file2, file3: FilePtr;
  folder1, folder2, folder3: FolderPtr;

begin
  // Create files
  file1 := NewFile('file1.txt', 100);
  file2 := NewFile('file2.txt', 200);
  file3 := NewFile('file3.txt', 300);

  // Create folders
  folder1 := NewFolder('folder1');
  folder2 := NewFolder('folder2');
  folder3 := NewFolder('folder3');

  // Add files to folder1
  folder1^.Add(file1);
  folder1^.Add(file2);

  // Add folder1 and file3 to folder2
  folder2^.Add(folder1);
  folder2^.Add(file3);

  // Add folder2 to folder3
  folder3^.Add(folder2);

  // Print the initial folder structure
  writeln('Initial Folder Structure:');
  writeln(folder3^.AsString); writeln;

  // Delete file2 from folder1
```

```
  folder1^.Delete('file2.txt');

  // Print the updated folder structure after removing file2
  writeln('Folder Structure after Removing file2:');
  writeln(folder3^.AsString); writeln;

  // Delete folder1 from folder2
  folder2^.Delete('folder1');

  // Print the updated folder structure after deleting folder1
  writeln('Folder Structure after Deleting folder1:');
  writeln(folder3^.AsString); writeln;

  // Delete file3 from folder2
  folder2^.Delete('file3.txt');

  // Print the updated folder structure after removing file3
  writeln('Folder Structure after Removing file3:');
  writeln(folder3^.AsString); writeln;


  // Create folder1 and file1 again and add file1 to folder1
  folder1 := NewFolder('folder1');
  file1 := NewFile('file1.txt', 100);
  folder1^.Add(file1);

  writeln('Folder Structure after create folder1 and file1 again and add file1
to folder1: ');
  writeln(folder1^.AsString); writeln;

  // Move file1 from folder1 to folder2
  folder1^.Move('file1.txt', folder2);

  // Print the updated folder structure after moving file1
  writeln('Folder Structure after Moving file1:');
  writeln(folder3^.AsString); writeln;
  writeln(folder1^.AsString); writeln;

  // Delete the folder and file objects
  Dispose(folder3, Done);
  Dispose(folder1, Done);
end.
```

```
Initial Folder Structure:
name: folder3, type: folder, dateModified: 30/05/2023 20:32:06, childrenAmount:1, size:600, children:
  name: folder2, type: folder, dateModified: 30/05/2023 20:32:06, childrenAmount:2, size:600, children:
  name: folder1, type: folder, dateModified: 30

Folder Structure after Removing file2:
name: folder3, type: folder, dateModified: 30/05/2023 20:32:06, childrenAmount:1, size:400, children:
  name: folder2, type: folder, dateModified: 30/05/2023 20:32:06, childrenAmount:2, size:400, children:
  name: folder1, type: folder, dateModified: 30

Folder Structure after Deleting folder1:
name: folder3, type: folder, dateModified: 30/05/2023 20:32:06, childrenAmount:1, size:300, children:
  name: folder2, type: folder, dateModified: 30/05/2023 20:32:06, childrenAmount:1, size:300, children:
  name: file3.txt, type: file, dateModified: 30

Folder Structure after Removing file3:
name: folder3, type: folder, dateModified: 30/05/2023 20:32:06, childrenAmount:1, size:0, children:
  name: folder2, type: folder, dateModified: 30/05/2023 20:32:06, childrenAmount:0, size:0, children:

Folder Structure after create folder1 and file1 again and add file1 to folder1:
name: folder1, type: folder, dateModified: 30/05/2023 20:32:06, childrenAmount:1, size:100, children:
  name: file1.txt, type: file, dateModified: 30/05/2023 20:32:06 ,size: 100

Folder Structure after Moving file1:
name: folder3, type: folder, dateModified: 30/05/2023 20:32:06, childrenAmount:1, size:100, children:
  name: folder2, type: folder, dateModified: 30/05/2023 20:32:06, childrenAmount:1, size:100, children:
  name: file1.txt, type: file, dateModified: 30

name: folder1, type: folder, dateModified: 30/05/2023 20:32:06, childrenAmount:0, size:0, children:

Heap dump by heaptrc unit of C:\Repos\2023SS_ADF\UE8\hu3\TestFS.exe
187 memory blocks allocated : 12632/13056
187 memory blocks freed     : 12632/13056
0 unfreed memory blocks : 0
True heap size : 196608 (96 used in System startup)
True free heap : 196512
```