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.