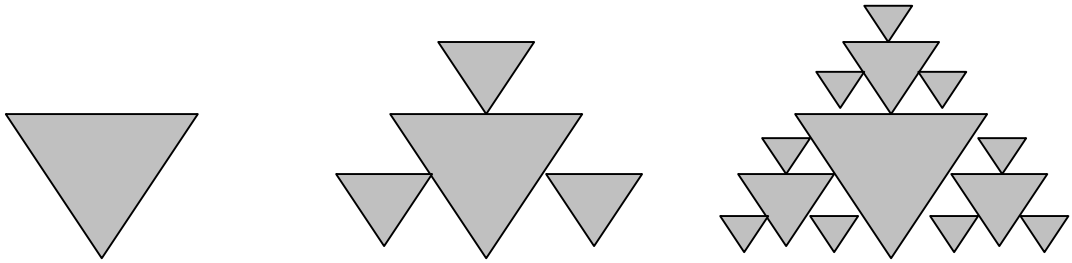
PRO1UEG*x* **Übungen zu Einführung in die Programmierung, Algorithmen und Datenstrukturen** **WS 25/26, Übung 7**

Abgabetermin: Mi, 07.01.2026

☒ **Gr. 1,** Dr. D. Auer **Name** Klemens Danner **Aufwand in h** 11
☐ **Gr. 2,** Dr. G. Kronberger
☐ **Gr. 3,** Dr. S. Wagner **Punkte** _____ **Kurzzeichen Tutor*in / Übungsleiter*in** ____ / ____

1. Ein Lichtlein brennt, ... dann vier, dann ... (4 + 2 Punkte)

Die Anzahl der Kerzen (*candles*), die man auf einem Christbaum unterbringen kann, hängt im Wesentlichen von der Höhe (*h*) des Baums ab. Studieren Sie folgende Beispiele von Christbaum-Beleuchtungen mittels Kerzen (ein Dreieck steht für die Flamme einer Kerze):



h = 1
candles = 1

h = 2
candles = 4

h = 3
candles = 13

a) Geben Sie eine *rekursive* Definition und einen *rekursiven* Algorithmus für *Candles*(*h*) an.

b) Geben Sie eine *iterative* Implementierung für *Candles*(*h*) an.

2. Die *Floor*-Funktion (6 Punkte)

Gegeben ist ein *binärer Suchbaum* für `INTEGER`-Werte gemäß folgender Deklarationen.

```
TYPE
  TreeNodePtr = ^TreeNode;
  TreeNode = RECORD
    left, right: TreeNodePtr;
    data: INTEGER;
  END; (* TreeNodePtr *)
  TreePtr = TreeNodePtr;
```

Gesucht ist die Funktion

```
FUNCTION Floor(tree: TreePtr; x: INTEGER): TreeNodePtr;
```

die den Zeiger auf den Knoten mit dem größten Wert in der `data`-Komponente liefert, der nicht größer als der Parameterwert `x` ist.

3. Morsen mit Binärbaum (12 Punkte)

Entwerfen Sie ein Programm, das eine in Morsecode vorliegende Nachricht einliest und in Textform ausgibt. Die vorliegende Nachricht in Morsecode ist so aufgebaut, dass die Codes der einzelnen Zeichen durch ein Leerzeichen und Wörter durch einen Strichpunkt getrennt werden.

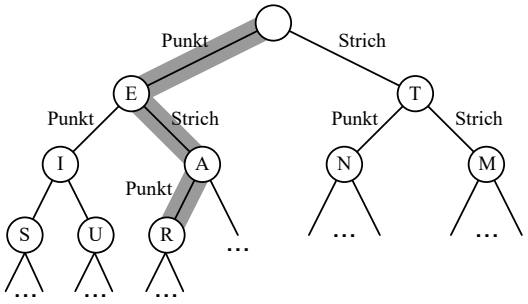
Beispiel (Nachricht):

```
... .;.... .- --. . -. -... . .-. --.
```

Ausgabe:

```
SE HAGENBERG
```

Für die Decodierung des Morsecodes möchten Sie einen Binärbaum einsetzen. Der Binärbaum ist folgendermaßen aufgebaut: Die Datenkomponente `ch` eines Knotens (außer der Wurzel) repräsentiert einen Buchstaben aus dem Alphabet A bis Z. Der Pfad von der Wurzel zu einem Knoten repräsentiert den Morsecode des Buchstabens, wobei die Verbindung zu einem linken Nachfolgeknoten einem Punkt und die zu einem rechten Nachfolger einem Strich entspricht. In der folgenden Darstellung ist beispielsweise der Pfad zu dem Buchstaben R mit dem Morsecode `.-.` hervorgehoben.



Entwickeln Sie eine Prozedur

```
PROCEDURE InsertMorseCode(tree: MorseTreePtr; ch: CHAR; code: STRING);
```

welche das Zeichen `ch` entsprechend der oben beschriebenen Baumorganisation in den Morsebaum `tree` einfügt. Rufen Sie diese Prozedur aus dem Hauptprogramm für alle Buchstaben A bis Z auf.

Hinweis: Sie können die Reihenfolge der Aufrufe frei wählen und so sicherstellen, dass für einen Aufruf (z. B. für Buchstaben R) die Knoten entlang des Pfades (z.B. E und A) bereits in dem Baum enthalten sind.

Entwickeln Sie dann eine Funktion

```
FUNCTION Lookup(tree: MorseTreePtr; code: STRING): CHAR;
```

welche für die Morsecodefolge `code` den im Baum `tree` gespeicherten Buchstaben liefert.

Testen Sie Ihr Programm und achten Sie auch auf Fehlerfälle, z.B. wenn die Nachricht ungültige Zeichen (z.B. `.?.`) oder ungültige Morsecodes (z.B. `.-.-.-.-`) enthält.

Morsecode:

```
A = .-      B = -...    C = -.-.
D = -..     E = .       F = ..-.    G = --.
H = ....    I = ..      J = .---    K = -.-
L = .-..    M = --      N = -.      O = ---
P = .--.    Q = --.-    R = .-.     S = ...
T = -       U = ..-     V = ...-    W = .--
X = -..-    Y = -.--    Z = --..
```

Hinweise:

1. Geben Sie für alle Ihre Lösungen immer eine „Lösungsidee“ an.
2. Dokumentieren und kommentieren Sie Ihre Pascal-Programme.
3. Geben Sie immer auch Testfälle ab, an denen man erkennen kann, dass Ihr Pascal-Programm funktioniert, und dass es auch in Fehlersituationen entsprechend reagiert.

1 Candles

1.1 Lösungsidee

Wenn die Höhe sich um 1 erhöht, scheinen sich alle Kerzen an jeder Seite der neuen "Hauptkerze" (die größte, mittlere in der Visualisierung) zu gruppieren. Da eine Kerze als Dreieck repräsentiert ist, verdreifacht sich die Anzahl der Kerzen, zusätzlich gibt es eine neue "Hauptkerze". Eine Christbaum der Höhe 1 hat genau eine Kerze. Damit ist die rekursive Lösung gefunden.

Die iterative Lösung funktioniert im Prinzip gleich. Die Variable result wird mit 1 initialisiert. Ist die Höhe gleich 1, so wird die Schleife übersprungen und 1 zurückgegeben. Ansonsten wird in einer Schleife bis h hochgezählt und immer das vorläufige Ergebnis verdreifacht und 1 addiert.

1.2 Quellcode

```
PROGRAM Candles;

  FUNCTION CandlesRec(h: INTEGER): INTEGER;
  BEGIN
    IF h = 1 THEN BEGIN
      CandlesRec := 1;
    END ELSE BEGIN
      CandlesRec := 1 + (3 * CandlesRec(h-1));
    END;
  END;

  FUNCTION CandlesIt(h: INTEGER): INTEGER;
    VAR
      i: INTEGER;
      result: INTEGER;
  BEGIN
    result := 1;
    FOR i := 2 TO h DO BEGIN
      result := 1 + (3 * result);
    END;
    CandlesIt := result;
  END;

  VAR
    userInput: INTEGER;

BEGIN
  Write('How high is the tree? > ');
  ReadLn(userInput);

  IF userInput > 0 THEN BEGIN
    Write('CandlesRec:');
    WriteLn(CandlesRec(userInput):3);

    Write('CandlesIt:');
    WriteLn(CandlesIt(userInput):3);
  END ELSE BEGIN
    WriteLn('Enter a positive value');
  END;
END.
```

1.3 Tests

Input	Beschreibung	Output

1	-	`How high is the tree? > 1`
		`CandlesRec:  1`
		`CandlesIt:  1`

2	-	`How high is the tree? > 2`
		`CandlesRec:  4`
		`CandlesIt:  4`

3	-	`How high is the tree? > 3`
		`CandlesRec: 13`
		`CandlesIt: 13`

0	Wert kleiner gleich 1	`How high is the tree? > 0` `Enter a positive value`

2 Floor-Funktion

2.1 Lösungsidee

Der Baum soll iterativ so lange durchwandert werden, bis entweder genau der gesuchte Wert gefunden ist, oder einer der Teilbäume NIL ist. In der Schleife wird jeder Wert als potenzieller Treffer gespeichert, der kleiner als der gegebene Wert x ist.
Ist der Zeiger auf den nächsten Knoten (bei mir "st") gleich dem Wert x, wird dieser zurückgegeben, ansonsten der zuletzt gespeicherte Treffer.
Hilfsfunktionen ermöglichen die Ein- und Ausgabe des Baumes.

2.2 Quellcode

```
PROGRAM FindFloor;
  TYPE
    TreeNodePtr = ^TreeNode;
    TreeNode = RECORD
      left, right: TreeNodePtr;
      data: INTEGER;
    END;
    TreePtr = TreeNodePtr;

  PROCEDURE InsertTree(VAR tree: TreePtr; val: INTEGER);
    VAR
      n: TreeNodePtr;
      st, pt: TreeNodePtr;
  BEGIN
    New(n);
    n^.left := NIL;
    n^.right := NIL;
    n^.data := val;

    IF tree = NIL THEN BEGIN
      tree := n;
      Exit;
    END;

    st := tree;
    pt := NIL;
    WHILE st <> NIL DO BEGIN
      pt := st;
      IF val < st^.data THEN BEGIN
        st := st^.left;
      END ELSE BEGIN
        st := st^.right;
      END;
    END;

    IF val < pt^.data THEN BEGIN
      pt^.left := n;
    END ELSE BEGIN
      pt^.right := n;
    END;
  END;

  PROCEDURE ReadTree(VAR tree: TreePtr);
```

```
    VAR
      val: INTEGER;
  BEGIN
    val := -1;
    WHILE val <> 0 DO BEGIN
      Write(' > ');
      ReadLn(val);
      IF val <> 0 THEN BEGIN
        InsertTree(tree, val);
      END;
    END;
  END;

  PROCEDURE WriteTree(tree: TreePtr);
  BEGIN
    IF tree <> NIL THEN BEGIN
      WriteTree(tree^.left);
      Write(tree^.data:3);
      WriteTree(tree^.right);
    END;
  END;

  PROCEDURE DisposeTree(VAR tree: TreePtr);
  BEGIN
    IF tree <> NIL THEN BEGIN
      DisposeTree(tree^.left);
      DisposeTree(tree^.right);
      Dispose(tree);
    END;
    tree := NIL;
  END;

  FUNCTION Floor(tree: TreePtr; x: INTEGER): TreeNodePtr;
    VAR
      st: TreeNodePtr;
      possResult, result: TreeNodePtr;
  BEGIN
    IF tree = NIL THEN BEGIN
      Floor := NIL;
      Exit;
    END;
    st := tree;
    possResult := NIL;
    result := NIL;

    WHILE (st <> NIL) AND (st^.data <> x) DO BEGIN
```

```
90          IF st^.data < x THEN BEGIN (*mögl. Treffer, rechts weitersuchen*)
91            possResult := st;
92            st := st^.right;
93          END ELSE BEGIN (* st^.data > x, links weitersuchen*)
94            st := st^.left;
95          END;
96        END;
97
98        IF st = NIL THEN BEGIN
99          result := possResult;
100       END ELSE BEGIN
101         result := st;
102       END;
103
104       Floor := result;
105     END;
106
107     VAR
108       t: TreePtr;
109       x: INTEGER;
110       tmp: TreeNodePtr;
111
112   BEGIN
113     t := NIL;
114     ReadTree(t);
115     WriteTree(t);
116     WriteLn;
117
118     Write(' Floor of > ');
119     ReadLn(x);
120     tmp := Floor(t, x);
121     IF tmp <> NIL THEN BEGIN
122       WriteLn(tmp^.data);
123     END ELSE BEGIN
124       WriteLn('Error: value invalid or tree empty');
125     END;
126
127     DisposeTree(t);
128   END.
```

2.3 Tests

Input	Begründung	Output
2 4 3 5 9 7 6 x = 8	Insert funktioniert und richtiger Floor bei Standard-Input	> 2 > 4 > 3 > 5 > 9 > 7 > 6 > 0 2 3 4 5 6 7 9 Floor of > 8 7
2 4 3 5 9 7 6 x = 5	Wert x kommt im Baum vor	2 3 4 5 6 7 9 Floor of > 5 5
-	Leerer Baum	> 0 Floor of > 4 Error: value invalid or tree empty
4 5 7 9 x = 3	Wert x kleiner als alle Werte im Baum, es gibt keinen Floor	> 4 > 5 > 7 > 9 > 0 4 5 7 9 Floor of > 3 Error: value invalid or tree empty
4 5 7 9 x = 10	Wert x größer als alle Werte im Baum	> 4 > 5 > 7 > 9 > 0 4 5 7 9 Floor of > 10 9

3 Morsen mit Binärbaum

3.1 Lösungsidee

Die Prozedur InsertMorseCode soll den angegebenen Code als "Wegweiser" verwenden:

- ist das nächste Zeichen im Code ein '.', wird links fortgeführt
- ist das nächste Zeichen im Code ein '-', wird links fortgeführt
  Damit die Reihenfolge der Aufrufe mit den Buchstaben egal ist, werden entlang dem Weg Dummyknoten angelegt, wenn die Knoten entlang eines Pfades noch nicht existieren. Alle diese Werte müssen am Ende mit echten Buchstaben überschrieben sein, was die Funktion TreeValid prüft.

Die Funktion Lookup folgt einem ähnlichen Prinzip, nur dass keine neuen Knoten eingefügt werden müssen. Der Code fungiert als "Wegweiser". Zurückgegeben wird der ch-Wert des Knotens, auf den der Zeiger 'st' nach der Schleife zeigt.

Die Funktion ConvertMessage übernimmt die finale Konvertierung des Codes in einen string. Der Hauptcode wird dabei Zeichen für Zeichen durchlaufen und dabei ein output zusammengebaut. Immer wenn ein Leerzeichen oder Strichpunkt kommt, endet ein Buchstabe, welcher mit Lookup übersetzt wird und zum output hinzugefügt wird. Bei einem Strichpunkt wird zusätzlich ein Leerzeichen hinzugefügt (Wortende).

3.2 Quellcode

```
1   PROGRAM MorseCode;
2
3     TYPE
4       TreeNodePtr = ^TreeNode;
5       TreeNode = RECORD
6         left, right: TreeNodePtr;
7         ch: CHAR;
8       END;
9       MorseTreePtr = TreeNodePtr;
10
11    PROCEDURE InsertMorseCode(tree: MorseTreePtr; ch: CHAR; code: STRING);
12      VAR
13        n, st: MorseTreePtr;
14        i: INTEGER;
15    BEGIN
16
17      IF tree = NIL THEN BEGIN
18        WriteLn('Error: Root Node does not exist');
19        Exit;
20      END;
21      tree^.ch := '/'; (*root*)
22
23      i := 1;
24      st := tree;
25      n := NIL;
26      WHILE i <= length(code) DO BEGIN
27        IF code[i] = '.' THEN BEGIN (*links gehen*)
28          IF st^.left = NIL THEN BEGIN (*Knoten erstellen, mit dummyWert
    füllen, in den Baum einhängen, damit man "drüberlaufen kann"*)
29            New(n);
30            n^.left := NIL;
31            n^.right := NIL;
32            n^.ch := ' ';
33            st^.left := n;
34          END;
35          st := st^.left;
36
37        END ELSE BEGIN
38          IF code[i] = '-' THEN BEGIN (*rechts gehen*)
39            IF st^.right = NIL THEN BEGIN
40              New(n);
41              n^.left := NIL;
42              n^.right := NIL;
```

```
43              n^.ch := ' ';
44              st^.right := n;
45            END;
46            st := st^.right;
47          END ELSE BEGIN (*ungültiger Code, Exit*)
48            WriteLn('Error: invalid Code - Exit');
49            Exit;
50          END;
51        END;
52        inc(i);
53      END;
54
55      IF i = 1 + length(code) THEN BEGIN (*st ist an der richigen Stelle,
    neuer Knoten bereits da, nur mehr Wert ändern*)
56        st^.ch := ch;
57      END ELSE BEGIN (*Fehler (i muss gleich 1+length(code) sein)*)
58        WriteLn('Error');
59        Exit;
60      END;
61    END;
62
63    FUNCTION Lookup(tree: MorseTreePtr; code: STRING): CHAR;
64      VAR
65        st: MorseTreePtr;
66        i: INTEGER;
67    BEGIN
68      i := 1;
69      st := tree;
70      WHILE (i <= length(code)) AND (st <> NIL) DO BEGIN
71        IF code[i] = '.' THEN BEGIN
72          st := st^.left;
73        END ELSE BEGIN
74          IF code[i] = '-' THEN BEGIN
75            st := st^.right;
76          END ELSE BEGIN
77            WriteLn('Error: invalid Code - Exit');
78            Lookup := '?';
79            Exit;
80          END;
81        END;
82        inc(i);
83      END;
84
85      IF st <> NIL THEN BEGIN
86        Lookup := st^.ch;
87      END ELSE BEGIN (* nicht enthalten*)
```

```
88        Lookup := '?';
89      END;
90    END;
91
92    FUNCTION ConvertMessage(tree: MorseTreePtr; mCode: STRING): STRING;
93      VAR
94        output, tmpCode: STRING;
95        i: INTEGER;
96    BEGIN
97      output := '';
98      IF (tree = NIL) OR (mCode = '') THEN BEGIN
99        ConvertMessage := 'Tree Empty or code missing';
100       Exit;
101     END;
102
103     i := 1;
104     tmpCode := '';
105     output := '';
106     WHILE i <= length(mCode) DO BEGIN
107       IF NOT (mCode[i] IN [' ', ';']) THEN BEGIN (*bei gültigem Code also
    nur . oder -, ungültige Zeichen zu tmpCode hinzugefügt und nachher an Lookup
    weitergegeben*)
108         tmpCode := tmpCode + mCode[i];
109         IF i = length(mCode) THEN BEGIN
110           output := output + Lookup(tree, tmpCode);
111         END;
112       END ELSE BEGIN (* bei gültigem code also nur ' ' oder ;*)
113         IF mCode[i] = ' ' THEN BEGIN
114           IF tmpCode <> '' THEN BEGIN (*Fall, dass zuerst ';' und
    darauffolgend ' ' eingegeben wird, wird ausgeschlossen*)
115             output := output + Lookup(tree, tmpCode);
116             tmpCode := '';
117           END;
118         END ELSE BEGIN
119           IF mCode[i] = ';' THEN BEGIN
120             IF tmpCode <> '' THEN BEGIN (*Fall, dass zuerst ' ' und
    darauffolgend ';' eingegeben wird, wird ausgeschlossen*)
121               output := output + Lookup(tree, tmpCode) + ' ';
122               tmpCode := '';
123             END;
124           END;
125         END;
126       END;
127       inc(i);
128     END;
129
```

```
130       ConvertMessage := output;
131     END;
132
133     FUNCTION TreeValid(tree: MorseTreePtr): BOOLEAN; (*prüft, dass keine
    Dummywerte mehr vorhanden sind*)
134     BEGIN
135       IF tree = NIL THEN BEGIN
136         TreeValid := TRUE;
137       END ELSE BEGIN
138         IF tree^.ch = ' ' THEN BEGIN
139           TreeValid := FALSE;
140         END ELSE BEGIN
141           TreeValid := TreeValid(tree^.left) AND TreeValid(tree^.right);
142         END;
143       END;
144     END;
145
146     PROCEDURE DisposeTree(VAR tree: MorseTreePtr);
147     BEGIN
148       IF tree <> NIL THEN BEGIN
149         DisposeTree(tree^.left);
150         DisposeTree(tree^.right);
151         Dispose(tree);
152         tree := NIL;
153       END;
154     END;
155
156     VAR
157       tree: MorseTreePtr;
158       mCode: STRING;
159
160   BEGIN
161     New(tree); (*root erstellen*)
162     tree^.right := NIL;
163     tree^.left := NIL;
164     tree^.ch := '/';
165
166     InsertMorseCode(tree, 'A', '.-');
167     InsertMorseCode(tree, 'B', '-...');
168     InsertMorseCode(tree, 'C', '-.-.');
169     InsertMorseCode(tree, 'D', '-..');
170     InsertMorseCode(tree, 'E', '.');
171     InsertMorseCode(tree, 'F', '..-.');
172     InsertMorseCode(tree, 'G', '--.');
173     InsertMorseCode(tree, 'H', '....');
174     InsertMorseCode(tree, 'I', '..');
```

```
    InsertMorseCode(tree, 'J', '.---');
    InsertMorseCode(tree, 'K', '-.-');
    InsertMorseCode(tree, 'L', '.-..');
    InsertMorseCode(tree, 'M', '--');
    InsertMorseCode(tree, 'N', '-.');
    InsertMorseCode(tree, 'O', '---');
    InsertMorseCode(tree, 'P', '.--.');
    InsertMorseCode(tree, 'Q', '--.-');
    InsertMorseCode(tree, 'R', '.-.');
    InsertMorseCode(tree, 'S', '...');
    InsertMorseCode(tree, 'T', '-');
    InsertMorseCode(tree, 'U', '..-');
    InsertMorseCode(tree, 'V', '...-');
    InsertMorseCode(tree, 'W', '.--');
    InsertMorseCode(tree, 'X', '-..-');
    InsertMorseCode(tree, 'Y', '-.--');
    InsertMorseCode(tree, 'Z', '--..');

    IF TreeValid(tree) THEN BEGIN
      Write('Enter a Code > ');
      ReadLn(mCode);
      WriteLn(ConvertMessage(tree, mCode));
    END ELSE BEGIN
      WriteLn('Tree invalid');
    END;

    DisposeTree(tree);

  END.
```

3.3 Tests

Input	Beschreibung	Output
Standard	Allgemeiner Test	`Enter a Code > ... .;.... .- --. . -. -... . .-. --.` `SE HAGENBERG`
-	leer lassen	`Enter a Code >` `Tree Empty or code missing`

leerer Baum (tree = nil)	Fehlermeldung muss kommen	`Error: Root Node does not exist`
		`Error: Root Node does not exist`
		`Error: Root Node does not exist`
		`Enter a Code > ...`
		`Tree Empty or code missing`

' ' folgt auf ;	Lookup wird nicht mit einem leeren Code aufgerufen	`Enter a Code > ... .; .... .- --. . -. -... . .-. --.` `SE HAGENBERG`
siehe Screenshot	gesuchter Knoten existiert im Baum nicht - liefert ein '?' für dasjenige Zeichen	`Enter a Code > ... .;.... .- --. . -. -... . .-.-.- --.` `SE HAGENBE?G`

andere Zeichen als Code	Erwartet: Fehlermeldung + Lookup gibt ein '?' zurück, wenn ein Wert nicht gefunden wird	`Enter a Code > ... .;.... .- --. . -a -... . .-. --.`
		`Error: invalid Code - Exit`
		`SE HAGE?BERG`

InsertMorseCode(tree, 'Z', '--...');	Hier ist beim Hinzufügen eines Buchstabens im Code ein Punkt zu viel. Es gibt also min einen Dummyknoten,	`Tree invalid`

Input	Beschreibung	Output
	der Baum ist invalid	