

<input type="checkbox"/> Gr. 1, Dr. G. Kronberger	Name <u>Andreas Roither</u>	Aufwand in h <u>6 h</u>
<input type="checkbox"/> Gr. 2, Dr. H. Gruber		
<input checked="" type="checkbox"/> Gr. 3, Dr. D. Auer	Punkte _____	Kurzzeichen Tutor / Übungsleiter _____ / _____

1. ADS: Behälter für Wörter

(6 Punkte)

Implementieren Sie einen Behälter (*container*) für Wörter (also für Werte des Datentyps *STRING*) auf Basis eines binären Suchbaums als abstrakte Datenstruktur (ADS) in Form eines Moduls (Pascal-UNIT) mit der Bezeichnung *WC_ADS* (für *word container as abstract data structure*).

Als Operationen müssen mindestens *IsEmpty*, *Insert*, *Remove* und *Contains* zur Verfügung gestellt werden.

2. ADT: Behälter für Wörter

(4 Punkte)

Realisieren Sie (auf Basis Ihrer Erfahrungen aus Aufgabe 1) einen abstrakten Datentyp (AT) für Wortbehälter (*word container*) auf Basis binärer Suchbäume in Form eines Moduls (Pascal-UNIT) mit der Bezeichnung *WC_ADT* (für *word container as abstract data type*).

Als Operationen müssen (wieder) mindestens *IsEmpty*, *Insert*, *Remove* und *Contains* zur Verfügung gestellt werden.

3. ADT: Menge

(14 Punkte)

Eine Menge (im Sinne der Mathematik) enthält jedes Element nur einmal, wobei die Reihenfolge der Elemente irrelevant ist. In einer Menge von Wörtern, einer Wortmenge, kommt also jedes Wort nur einmal vor. Somit gilt: { 'a', 'b', 'b', 'a' } = { 'a', 'b' } = { 'b', 'a' }.

Realisieren Sie (auf Basis Ihrer Erfahrungen aus Aufgabe 1 und 2) einen abstrakten Datentyp (ADT) für Wortmengen (*word sets*) in Form eines Moduls (Pascal-UNIT) mit der Bezeichnung *WS_ADT*.

Neben den schon aus den ersten Aufgaben bekannten Operationen *IsEmpty*, *Insert*, *Remove* und *Contains* müssen nun auch die typischen Mengenoperationen *Union*, *Intersection* und *Difference* in Form von Funktionen mit folgender Schnittstelle

```
FUNCTION ... (s1, s2: WordSet): WordSet;
```

realisiert werden, sowie die Operation *Cardinality*, welche die Anzahl der Elemente einer Menge liefert.

Um Ihre Mengenimplementierung zu testen, versuchen Sie zu überprüfen, ob die derzeitige Koalitionsform in Österreich (große Koalition aus SPÖ und ÖVP) wirklich eine gute Idee war. Lesen Sie dazu die Parteiprogramme der vier größeren, derzeit im Nationalrat vertretenen Parteien (SPÖ, ÖVP, FPÖ, GRÜNE, also ohne Team Stronach und NEOS, siehe *Parteiprogramme.zip* im Moodle-Kurs) jeweils in eine Wortmenge ein und bilden dann alle möglichen Schnittmengen aus jeweils zweien davon. Als Ausgangsbasis für Ihr Programm können Sie den Inhalt von *WordStuff.zip* verwenden.

... mal sehen, ob die Parteiprogramme von SPÖ und ÖVP wirklich die meisten "Gemeinsamkeiten" (also die größte Schnittmenge in Form von gleichen Wörtern) aufweisen, oder ob nicht eine andere Koalition (zumindest auf dieser rein textuellen Basis;-) besser geeignet wäre.

Übung 4

Aufgabe 1

Lösungsidee

Auf Basis eines binären Suchbaums wird eine Pascal-Unit erstellt. Diese Unit enthält folgende Operationen:

- IsEmpty:
Gibt True oder False zurück, je nachdem ob etwas in dem Baum enthalten ist oder nicht
- Insert:
Ermöglicht es etwas in den Baum einzufügen
- Remove:
Entfernt eine Node des Baumes
- Contains:
Gibt True oder False zurück, je nachdem ob ein bestimmter String im Baum enthalten ist

```
1 (* WC_ADS                23.04.2017 *)
2 (* Container for strings  *)
3
4
5 UNIT WC_ADS;
6
7 INTERFACE
8
9     FUNCTION IsEmpty: BOOLEAN;
10    FUNCTION Contains(s: STRING): BOOLEAN;
11    PROCEDURE Insert(s: STRING);
12    PROCEDURE Remove(s: STRING);
13    PROCEDURE DisposeTree();
14
15 IMPLEMENTATION
16
17 TYPE
18     Node = ^NodeRec;
19     NodeRec = RECORD
20         data: STRING;
21         left, right: Node;
22     END;
23
24 VAR
25     Tree : Node;
26
27 PROCEDURE InitTree;
28 BEGIN
29     Tree := NIL;
```

```
29  END;

31  FUNCTION NewNode (data: STRING): Node;
    VAR
33      n: Node;
    BEGIN
35      New(n);
        n^.data := data;
37      n^.left := NIL;
        n^.right := NIL;
39      NewNode := n;
    END;

41  (* Check if binsearchtree is empty *)
43  FUNCTION IsEmpty: BOOLEAN;
    BEGIN
45      IsEmpty := Tree = NIL;
    END;

47  (* check if binsearchtree contains string
49      recursive *)
    FUNCTION ContainsRec(VAR t: Node; s: STRING): BOOLEAN;
    BEGIN
51      IF t = NIL THEN BEGIN
53          ContainsRec := FALSE;
          END
55      ELSE IF t^.data = s THEN
          ContainsRec := TRUE
57      ELSE IF s < t^.data THEN BEGIN
          ContainsRec := ContainsRec(t^.left, s);
59      END
          ELSE BEGIN
61          ContainsRec := ContainsRec(t^.right, s);
          END;
63      END;

65  (* check if binsearchtree contains string
        Uses a help function *)
67  FUNCTION Contains(s: STRING): BOOLEAN;
    BEGIN
69      Contains := ContainsRec(Tree, s);
    END;

71  (* Insert in binsearchtree
73      recursive *)
    PROCEDURE InsertRec (VAR t: Node; n: Node);
75  BEGIN
        IF t = NIL THEN BEGIN
```

```

77     t := n;
      END
79   ELSE BEGIN
      IF (n^.data = t^.data) THEN Exit
81   ELSE IF n^.data < t^.data THEN
      InsertRec(t^.left, n)
83   ELSE
      InsertRec(t^.right, n)
85   END;
    END;

87   (* Insert a string in binsearchtree
88   Uses a help function *)
    PROCEDURE Insert (s : String);
91   BEGIN
      InsertRec(Tree, NewNode(s));
93   END;

95   (* Remove a string from binsearchtree *)
    PROCEDURE Remove(s: STRING);
97   VAR
      n, nPar: Node;
99   st: Node; (*subtree*)
      succ, succPar: Node;
101  BEGIN
      nPar := NIL;
103   n := Tree;
    WHILE (n <> NIL) AND (n^.data <> s) DO BEGIN
105   nPar := n;
      IF s < n^.data THEN
107   n := n^.left
      ELSE
109   n := n^.right;
      END;
111   IF n <> NIL THEN BEGIN (* no right subtree *)
      IF n^.right = NIL THEN BEGIN
113   st := n^.left;
      END
115   ELSE BEGIN
      IF n^.right^.left = NIL THEN BEGIN (* right subtree, but no left subtree *)
117   st := n^.right;
      st^.left := n^.left;
119   END
      ELSE BEGIN
121   (*common case*)
      succPar := NIL;
      succ := n^.right;
123   WHILE succ^.left <> NIL DO BEGIN

```

```

125     succPar := succ;
126     succ := succ^.left;
127 END;
128     succPar^.left := succ^.right;
129     st := succ;
130     st^.left := n^.left;
131     st^.right := n^.right;
132 END;
133 END;
134 (* insert the new sub-tree *)
135 IF nPar = NIL THEN
136     Tree := st
137 ELSE IF n^.data < nPar^.data THEN
138     nPar^.left := st
139 ELSE
140     nPar^.right := st;
141 Dispose(n);
142 END; (* n <> NIL *)
143 END;

144 (* Removes all the elements from the binary search tree
145    recursive *)
146 PROCEDURE DisposeTree_rec(VAR Tree : Node);
147 BEGIN
148     IF Tree <> NIL THEN BEGIN
149
150         (* Traverse the left subtree in postorder. *)
151         DisposeTree_rec (Tree^.Left);
152
153         (* Traverse the right subtree in postorder. *)
154         DisposeTree_rec (Tree^.Right);
155
156         (* Delete this leaf node from the tree. *)
157         Dispose (Tree);
158     END
159 END;

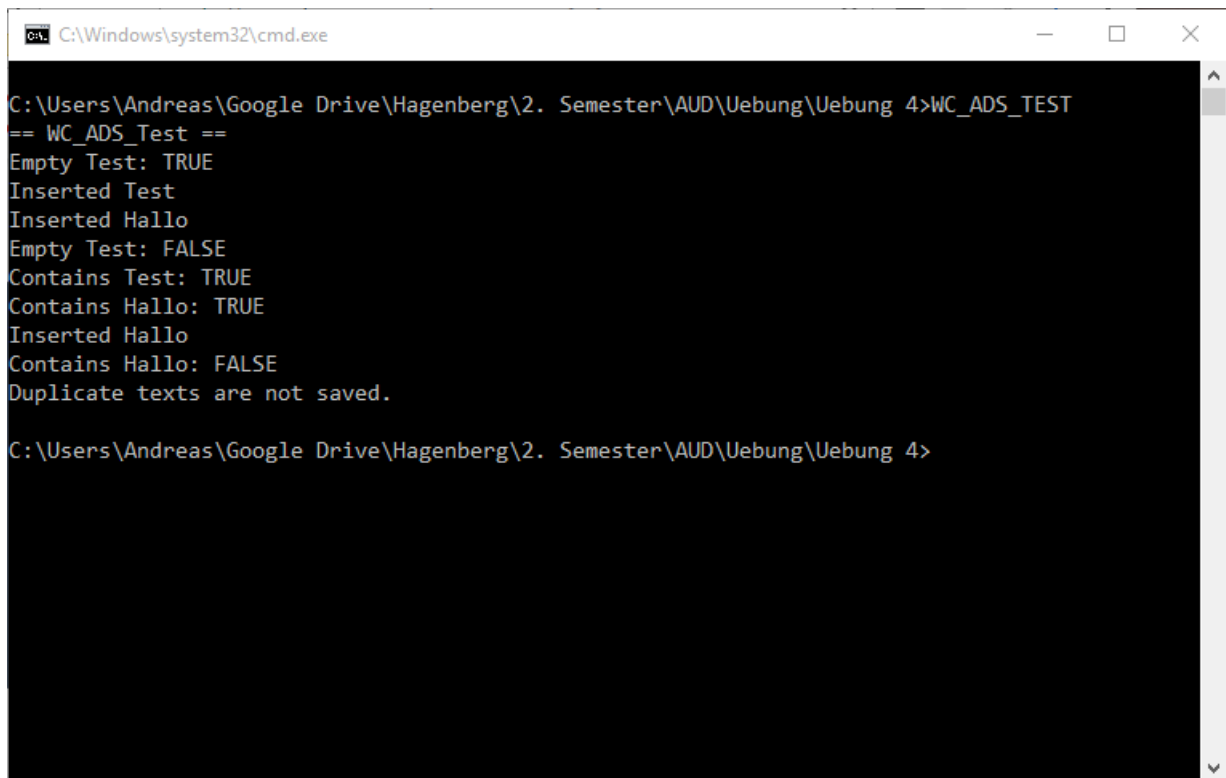
160 (* Removes all the elements from the binary search tree
161    calls rec. function *)
162 PROCEDURE DisposeTree();
163 BEGIN
164     DisposeTree_rec(Tree);
165 END;

166 BEGIN
167     InitTree;
168 END.

```

```
1  (* WC_ADT_TEST           23.04.2017 *)
2  (* Container for strings           *)
3
4  PROGRAM WC_ADS_Test;
5
6      USES WC_ADS;
7
8  BEGIN
9      (* test cases *)
10     WriteLn('== WC_ADS_Test ==');
11     WriteLn('Empty Test: ', IsEmpty());
12     WriteLn('Inserted Test');
13     Insert('Test');
14
15     WriteLn('Inserted Hallo');
16     Insert('Hallo');
17
18     WriteLn('Empty Test: ', IsEmpty());
19     WriteLn('Contains Test: ', Contains('Test'));
20     WriteLn('Contains Hallo: ', Contains('Hallo'));
21
22     WriteLn('Inserted Hallo');
23     Insert('Hallo');
24     Remove('Hallo');
25     WriteLn('Contains Hallo: ', Contains('Hallo'));
26     WriteLn('Duplicate texts are not saved. ');
27
28     DisposeTree;
29 END.
```

WC_ADS_TEST.pas



```
C:\Windows\system32\cmd.exe

C:\Users\Andreas\Google Drive\Hagenberg\2. Semester\AUD\Uebung\Uebung 4>WC_ADS_TEST
== WC_ADS_Test ==
Empty Test: TRUE
Inserted Test
Inserted Hallo
Empty Test: FALSE
Contains Test: TRUE
Contains Hallo: TRUE
Inserted Hallo
Contains Hallo: FALSE
Duplicate texts are not saved.

C:\Users\Andreas\Google Drive\Hagenberg\2. Semester\AUD\Uebung\Uebung 4>
```

Abbildung 1: WC ADS Unit Test

Die Testfälle zeigen die Verwendung der Funktionen. Bereits vorhandene Wörter werden nicht nochmal eingefügt, deshalb ist trotz neu einfügen des Wortes “Hallo“ und anschließendem löschen und abfragen ob es existiert, “Hallo“ nicht mehr vorhanden. Anders als bei Aufgabe 2 gibt es hier keinen Datentyp auf den zugegriffen werden kann.

Aufgabe 2

Lösungsidee

Genau wie Aufgabe 1 nur mit dem Unterschied das ein Typ vorhanden ist auf den zugegriffen werden kann. In dieser Implementation sind die Hintergrund Daten, also "Node", nicht sichtbar. Die Operationen funktionieren genau gleich für den Anwender mit der Ausnahme, das ein Objekt mit übergeben werden muss. In der Unit werden TypeCasts bei den Operationen ausgeführt damit der übergebene Pointer verwendet werden kann. Falls also ein anderer Pointer übergeben wird, der auf eine andere Datenstruktur als eine Node zeigt, wird es zu Laufzeitfehlern kommen. Jedoch wird somit sicher gestellt das von außerhalb der Unit nicht auf interne Daten zugegriffen werden kann.

```

1  (* WC_ADT                23.04.2017 *)
2  (* Container for strings      *)
3  (* Hidden data                *)

5  UNIT WC_ADT;

7  INTERFACE
8      TYPE
9          (* "hiding" of tree data *)
10         TreePtr = POINTER;

11
12         PROCEDURE InitTree(VAR Tree: TreePtr);
13         FUNCTION IsEmpty(VAR Tree: TreePtr): BOOLEAN;
14         FUNCTION Contains(VAR Tree: TreePtr; s: STRING): BOOLEAN;
15         PROCEDURE Insert (VAR Tree: TreePtr; s : String);
16         PROCEDURE Remove(VAR Tree: TreePtr; s: STRING);
17         PROCEDURE DisposeTree(VAR Tree : TreePtr);

19  IMPLEMENTATION

21      TYPE
22          Node = ^NodeRec;
23          NodeRec = RECORD
24              data: STRING;
25              left, right: Node;
26          END;

27
28      (* Init tree *)
29      PROCEDURE InitTree(VAR Tree: TreePtr);
30      BEGIN
31          Node(Tree) := NIL;
32      END;

33
34      (* create a new node
35       returns: Node *)
36      FUNCTION NewNode (data: STRING): Node;

```



```

37  VAR
    n: Node;
39  BEGIN
    New(n);
41  n^.data := data;
    n^.left := NIL;
43  n^.right := NIL;
    NewNode := n;
45  END;

47  (* Check if binsearchtree is empty *)
    FUNCTION IsEmpty(VAR Tree: TreePtr): BOOLEAN;
49  BEGIN
    IsEmpty := Node(Tree) = NIL;
51  END;

53  (* check if binsearchtree contains string
    recursive *)
55  FUNCTION ContainsRec(VAR t: TreePtr; s: STRING): BOOLEAN;
    BEGIN
57  IF t = NIL THEN BEGIN
    ContainsRec := FALSE;
59  END
    ELSE IF Node(t)^.data = s THEN
61  ContainsRec := TRUE
    ELSE IF s < Node(t)^.data THEN BEGIN
63  ContainsRec := ContainsRec(Node(t)^.left, s);
    END
65  ELSE BEGIN
    ContainsRec := ContainsRec(Node(t)^.right, s);
67  END;
    END;

69  (* check if binsearchtree contains string
    Uses a help function *)
71  FUNCTION Contains(VAR Tree: TreePtr; s: STRING): BOOLEAN;
73  BEGIN
    Contains := ContainsRec(Node(Tree), s);
75  END;

77  (* Insert in binsearchtree
    recursive *)
79  PROCEDURE InsertRec (VAR t: Node; n: Node);
    BEGIN
81  IF t = NIL THEN BEGIN
    t := n;
83  END
    ELSE BEGIN

```

```

85  IF (n^.data = t^.data) THEN Exit
    ELSE IF n^.data < t^.data THEN
87      InsertRec(t^.left, n)
    ELSE
89      InsertRec(t^.right, n)
    END;
91 END;

93 (* Insert a string in binsearchtree
    Uses a help function *)
95 PROCEDURE Insert (VAR Tree: TreePtr; s : String);
    BEGIN
97     InsertRec(Node(Tree), NewNode(s));
    END;

99
100 (* Remove a string from binsearchtree *)
101 PROCEDURE Remove(VAR Tree: TreePtr; s: STRING);
    VAR
103     n, nPar: Node;
    st: Node; (*subtree*)
105     succ, succPar: Node;
    BEGIN
107     nPar := NIL;
    n := Node(Tree);
109     WHILE (n <> NIL) AND (n^.data <> s) DO BEGIN
    nPar := n;
111     IF s < n^.data THEN
    n := n^.left
113     ELSE
    n := n^.right;
115     END;
    IF n <> NIL THEN BEGIN (* no right subtree *)
117     IF n^.right = NIL THEN BEGIN
    st := n^.left;
119     END
    ELSE BEGIN
121     IF n^.right^.left = NIL THEN BEGIN (* right subtree, but no left subtree *)
    st := n^.right;
123     st^.left := n^.left;
    END
    ELSE BEGIN
125     (*common case*)
    succPar := NIL;
    succ := n^.right;
127     WHILE succ^.left <> NIL DO BEGIN
    succPar := succ;
129     succ := succ^.left;
131     END;
    END;

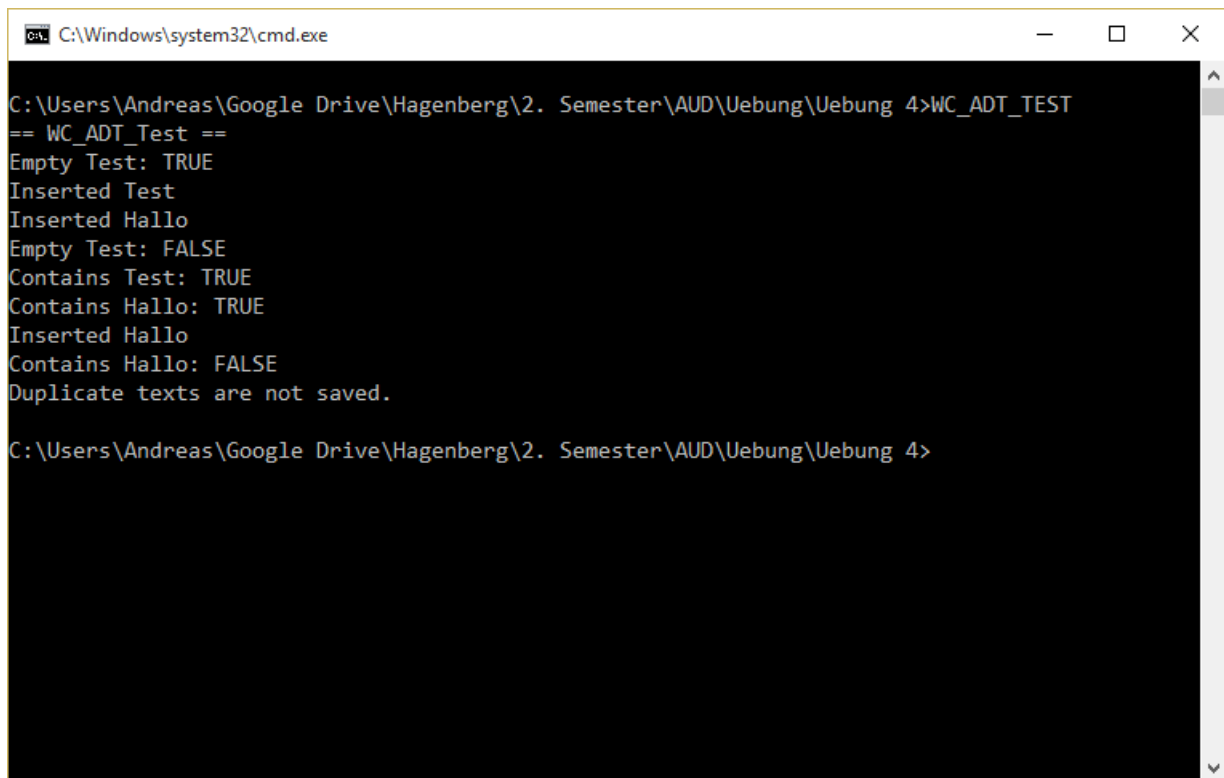
```

```
133      succPar^.left := succ^.right;
      st := succ;
135      st^.left := n^.left;
      st^.right := n^.right;
137      END;
      END;
139      (* insert the new sub-tree *)
      IF nPar = NIL THEN
141          Node(Tree) := st
      ELSE IF n^.data < nPar^.data THEN
143          nPar^.left := st
      ELSE
145          nPar^.right := st;
          Dispose(n);
147      END; (* n <> NIL *)
      END;
149
      (* Removes all the elements from the binary search tree *)
      (* rooted at Tree, leaving the tree empty. *)
      PROCEDURE DisposeTree(VAR Tree : TreePtr);
153      BEGIN
          (* Base Case: If Tree is NIL, do nothing. *)
155          IF Tree <> NIL THEN BEGIN
              (* Traverse the left subtree in postorder. *)
157              DisposeTree (Node(Tree)^.Left);
              (* Traverse the right subtree in postorder. *)
159              DisposeTree (Node(Tree)^.Right);
              (* Delete this leaf node from the tree. *)
163              Dispose (Node(Tree));
              END
165          END;
167      BEGIN
          END.
```

WC_ADT.pas

```
1  (* WC_ADT_TEST          23.04.2017 *)
2  (* Container for strings      *)
3  (* Hidden data test          *)
4
5  PROGRAM WC_ADT_Test;
6
7      USES WC_ADT;
8
9      VAR f1: POINTER;
10 BEGIN
11     InitTree(f1);
12
13     (* test cases *)
14     WriteLn('== WC_ADT_Test ==');
15     WriteLn('Empty Test: ', IsEmpty(f1));
16     WriteLn('Inserted Test');
17     Insert(f1, 'Test');
18
19     WriteLn('Inserted Hallo');
20     Insert(f1, 'Hallo');
21
22     WriteLn('Empty Test: ', IsEmpty(f1));
23     WriteLn('Contains Test: ', Contains(f1, 'Test'));
24     WriteLn('Contains Hallo: ', Contains(f1, 'Hallo'));
25
26     WriteLn('Inserted Hallo');
27     Insert(f1, 'Hallo');
28     Remove(f1, 'Hallo');
29     WriteLn('Contains Hallo: ', Contains(f1, 'Hallo'));
30     WriteLn('Duplicate texts are not saved. ');
31
32     DisposeTree(f1);
33 END.
```

WC_ADT_TEST.pas



```
C:\Windows\system32\cmd.exe

C:\Users\Andreas\Google Drive\Hagenberg\2. Semester\AUD\Uebung\Uebung 4>WC_ADT_TEST
== WC_ADT_Test ==
Empty Test: TRUE
Inserted Test
Inserted Hallo
Empty Test: FALSE
Contains Test: TRUE
Contains Hallo: TRUE
Inserted Hallo
Contains Hallo: FALSE
Duplicate texts are not saved.

C:\Users\Andreas\Google Drive\Hagenberg\2. Semester\AUD\Uebung\Uebung 4>
```

Abbildung 2: WC ADT Unit Test

Zu sehen sind genau die selben Tests wie bei Aufgabe 1. Da sich nur der Zugriff auf die Operationen geändert hat ist hier dasselbe Ergebnis zu sehen. Durch verwenden eines Datentyps können mehrere Objekte existieren die dieselben Funktionen haben. Dadurch können sich zwei Objekte vom Inhalt her unterscheiden. Bei Aufgabe 1 existiert nur ein Objekt innerhalb der Unit. Falls neue Daten eingespielt werden sollen, muss eine Lösch-Operation durchgeführt werden, erst dann können neue Daten eingefügt werden. Bei der Datentyp Variante wird einfach ein neues Objekt erstellt.

Aufgabe 3

Lösungsidee

Ähnlich wie bei Aufgabe 2 werden hier die Daten von Zugriff außerhalb der Unit geschützt. Zusätzlich werden noch weitere Operationen eingefügt:

- Union:
Gibt WordSet zurück das alle Nodes (außer Duplikaten) beider WordSets enthält
- Intersection:
Gibt WordSet zurück das Wörter enthält die in beiden übergebenen WordSets sind
- Difference:
Gibt WordSet zurück das Wörter enthält die nicht in beiden übergebenen WordSets sind
- Cardinality:
Gibt Anzahl der Elemente im Baum zurück

```

1  (* WS_ADT          23.04.2017 *)
2  (* Container for strings *)
3  (* Hidden data      *)
4
5  UNIT WS_ADT;
6
7  INTERFACE
8      TYPE
9          (* "hiding" of tree data *)
10         TreePtr = POINTER;
11
12         PROCEDURE InitTree(VAR Tree: TreePtr);
13         FUNCTION IsEmpty(VAR Tree: TreePtr): BOOLEAN;
14         FUNCTION Contains(VAR Tree: TreePtr; s: STRING): BOOLEAN;
15         PROCEDURE Insert (VAR Tree: TreePtr; s : String);
16         PROCEDURE Remove(VAR Tree: TreePtr; s: STRING);
17
18         FUNCTION Union (s1,s2 : TreePtr) : TreePtr;
19         FUNCTION Intersection (s1,s2 : TreePtr) : TreePtr;
20         FUNCTION Difference (s1,s2 : TreePtr) : TreePtr;
21         FUNCTION Cardinality (s1: TreePtr) : INTEGER;
22
23         PROCEDURE DisposeWS(VAR Tree: TreePtr);
24
25  IMPLEMENTATION
26
27      TYPE
28          WordSet = ^WordSetRec;
29          WordSetRec = RECORD
30              data: STRING;
31              left, right: WordSet;

```

```

33  END;
35  (* Init tree *)
36  PROCEDURE InitTree(VAR Tree: TreePtr);
37  BEGIN
38      WordSet(Tree) := NIL;
39  END;
41  (* create a new WordSet
42   returns: WordSet *)
43  FUNCTION NewWordSet (data: STRING): WordSet;
44  VAR
45      n: WordSet;
46  BEGIN
47      New(n);
48      n^.data := data;
49      n^.left := NIL;
50      n^.right := NIL;
51      NewWordSet := n;
52  END;
54  (* Check if binsearchtree is empty *)
55  FUNCTION IsEmpty(VAR Tree: TreePtr): BOOLEAN;
56  BEGIN
57      IsEmpty := WordSet(Tree) = NIL;
58  END;
59  (* check if binsearchtree contains string
60   TypeCast to WordSet for rec. function *)
61  FUNCTION ContainsRec(VAR t: TreePtr; s: STRING): BOOLEAN;
62  BEGIN
63      IF t = NIL THEN BEGIN
64          ContainsRec := FALSE;
65      END
66      ELSE IF WordSet(t).data = s THEN
67          ContainsRec := TRUE
68      ELSE IF s < WordSet(t).data THEN BEGIN
69          ContainsRec := ContainsRec(WordSet(t).left, s);
70      END
71      ELSE BEGIN
72          ContainsRec := ContainsRec(WordSet(t).right, s);
73      END;
74  END;
75  (* check if binsearchtree contains string
76   TypeCast to WordSet for rec. function *)
77  FUNCTION Contains(VAR Tree: TreePtr; s: STRING): BOOLEAN;
78  BEGIN
79

```

```

Contains := ContainsRec(WordSet(Tree), s);
81 END;

83 (* Insert in binsearchtree
    recursive *)
85 PROCEDURE InsertRec (VAR t: WordSet; n: WordSet);
BEGIN
87   IF t = NIL THEN BEGIN
        t := n;
89   END
    ELSE BEGIN
91   IF (n^.data = t^.data) THEN Exit
    ELSE IF n^.data < t^.data THEN
93     InsertRec(t^.left, n)
    ELSE
95     InsertRec(t^.right, n)
    END;
97 END;

99 (* Insert a string in binsearchtree
    TypeCast to WordSet for rec. function *)
101 PROCEDURE Insert (VAR Tree: TreePtr; s : String);
BEGIN
103   InsertRec(WordSet(Tree), NewWordSet(s));
END;

105
107 (* Remove a string from binsearchtree *)
PROCEDURE Remove(VAR Tree: TreePtr; s: STRING);
VAR
109   n, nPar: WordSet;
    st: WordSet; (*subtree*)
111   succ, succPar: WordSet;
BEGIN
113   nPar := NIL;
    n := WordSet(Tree);
115 WHILE (n <> NIL) AND (n^.data <> s) DO BEGIN
    nPar := n;
117   IF s < n^.data THEN
        n := n^.left
119   ELSE
        n := n^.right;
121   END;
    IF n <> NIL THEN BEGIN (* no right subtree *)
123     IF n^.right = NIL THEN BEGIN
        st := n^.left;
125     END
    ELSE BEGIN
127     IF n^.right^.left = NIL THEN BEGIN (* right subtree, but no left subtree *)

```



```

129     st := n^.right;
130     st^.left := n^.left;
131 END
132 ELSE BEGIN
133     (*common case*)
134     succPar := NIL;
135     succ := n^.right;
136     WHILE succ^.left <> NIL DO BEGIN
137         succPar := succ;
138         succ := succ^.left;
139     END;
140     succPar^.left := succ^.right;
141     st := succ;
142     st^.left := n^.left;
143     st^.right := n^.right;
144 END;
145 END;
146 (* insert the new sub-tree *)
147 IF nPar = NIL THEN
148     WordSet(Tree) := st
149 ELSE IF n^.data < nPar^.data THEN
150     nPar^.left := st
151 ELSE
152     nPar^.right := st;
153     Dispose(n);
154 END; (* n <> NIL *)
155 END;
156 (* copy a set
157    returns: WordSet*)
158 FUNCTION CopyWSet(ws : WordSet) : WordSet;
159 VAR
160     n : WordSet;
161 BEGIN
162     IF ws = NIL THEN CopyWSet := NIL
163     ELSE BEGIN
164         New(n);
165         n^.data := ws^.data;
166         n^.left := CopyWSet(ws^.left);
167         n^.right := CopyWSet(ws^.right);
168         CopyWSet := n;
169     END;
170 END;
171 (* inserts s1 into s2 and returns it
172    recursive *)
173 FUNCTION UnionRec (s1,s2 : WordSet) : WordSet;
174 VAR

```

```

    result : WordSet;
177 BEGIN
    result := s2;
179 IF s1 <> NIL THEN BEGIN
    Insert(result, s1^.data);
181 UnionRec(s1^.left, result);
    UnionRec(s1^.right, result);
183 END;
    UnionRec := result;
185 END;

187 (* union, calls recursive function
    TypeCast to WordSet for rec. function *)
189 FUNCTION Union (s1,s2 : TreePtr) : TreePtr;
BEGIN
191 WordSet(s1) := CopyWSet(WordSet(s1));
    WordSet(s2) := CopyWSet(WordSet(s2));
193 Union := WordSet(UnionRec(WordSet(s1),WordSet(s2)));
END;

195 (* removes Nodes that are not present in s2
    recursive *)
197 FUNCTION IntersectionRec (VAR s1,s2 : WordSet) : WordSet;
199 BEGIN
    IF s1 <> NIL THEN BEGIN
201 (* words that are NOT in both *)
        IF NOT Contains(s2 ,s1^.data) THEN BEGIN
203 Remove(s1, s1^.data);
            s1 := IntersectionRec(s1,s2);
205 END
        ELSE BEGIN
207 IntersectionRec(s1^.left, s2);
            IntersectionRec(s1^.right, s2);
209 END;
        END;
211 IntersectionRec := s1;
END;

213 (* intersection, calls recursive function
    TypeCast to WordSet for rec. function *)
215 FUNCTION Intersection (s1,s2 : TreePtr) : TreePtr;
217 BEGIN
    WordSet(s1) := CopyWSet(WordSet(s1));
219 WordSet(s2) := CopyWSet(WordSet(s2));
    Intersection := WordSet(IntersectionRec(WordSet(s1),WordSet(s2)));
221 END;

223 (* removes Nodes that are present in s2

```

```

    recursive *)
225 FUNCTION DifferenceRec (VAR s1,s2 : WordSet) : WordSet;
BEGIN
227   IF s1 <> NIL THEN BEGIN
       IF Contains(s2 ,s1^.data) THEN BEGIN
229         Remove(s1, s1^.data);
         s1 := DifferenceRec(s1,s2);
231       END
       ELSE BEGIN
233         DifferenceRec(s1^.left, s2);
         DifferenceRec(s1^.right, s2);
235       END;
       END;
237   DifferenceRec := s1;
END;

239 (* difference, calls recursive function
    TypeCast to WordSet for rec. function *)
241 FUNCTION Difference (s1,s2 : TreePtr) : TreePtr;
243 BEGIN
    WordSet(s1) := CopyWSet(WordSet(s1));
245   WordSet(s2) := CopyWSet(WordSet(s2));
    IF Cardinality(WordSet(s1)) < Cardinality(WordSet(s2)) THEN
247     Difference := DifferenceRec(WordSet(s2),WordSet(s1))
    ELSE
249     WordSet(Difference) := DifferenceRec(WordSet(s1),WordSet(s2));
END;

251 (* number of words in a WordSet
    returns 0 if none are found *)
253 FUNCTION Cardinality (s1: TreePtr) : INTEGER;
255 BEGIN
    (* 0 if nothing is found *)
257   Cardinality := 0;
    IF s1 <> NIL THEN BEGIN
259     Cardinality := 1;
     IF (WordSet(s1)^.left <> NIL) AND (WordSet(s1)^.right <> NIL) THEN BEGIN
261       Cardinality := Cardinality(WordSet(s1)^.left) + Cardinality(WordSet(s1)^.right) + 1;
     END
     ELSE
263       IF WordSet(s1)^.left <> NIL THEN Cardinality := Cardinality(WordSet(s1)^.left) +
        1
        ELSE IF WordSet(s1)^.right <> NIL THEN Cardinality := Cardinality(WordSet(s1)
        ^.right) + 1;
     END;
265   END;
267 END;

269 (* Removes all the elements from the binary search tree *)

```

```

271  (* rooted at Tree, leaving the tree empty. *)
PROCEDURE DisposeWS(VAR Tree : TreePtr);
BEGIN
273  (* Base Case: If Tree is NIL, do nothing. *)
  IF Tree <> NIL THEN BEGIN
275
    (* Traverse the left subtree in postorder. *)
277    DisposeWS (WordSet(Tree)^.Left);

    (* Traverse the right subtree in postorder. *)
279    DisposeWS (WordSet(Tree)^.Right);

281    (* Delete this leaf node from the tree. *)
283    Dispose (WordSet(Tree));
    END
285  END;
BEGIN
287  END.

```

WS_ADT.pas

```

1  (* WS_ADT_TEST          23.04.2017 *)
   (* Container for strings      *)
3  (* Hidden data test          *)

5  PROGRAM WS_ADT_TEST;

7  USES WS_ADT, WordReader;

9  (* Load words from files
   uses WordReader unit *)
11 PROCEDURE LoadFromFile(source: STRING; VAR ws: POINTER);
   VAR word: STRING;
13 BEGIN
    InitTree(ws);
15    word := '';

17    OpenFile(source, noConversion);
    ReadWord(word);

19
    (* ReadWord returns '' if EOF *)
21    WHILE (word <> '') DO BEGIN
        Insert(ws,word);
23        ReadWord(word);
    END;
25    CloseFile;
    END;

27  (* since WordSet is hidden, POINTER is used

```

```

29   WS_ADT interprets it internally as WordSet *)
    VAR f1, f2 ,f3, f4 : POINTER;
31 BEGIN

33   (* Load files *)
    LoadFromFile('gruene.txt', f1);
35   LoadFromFile('oep.txt', f2);
    LoadFromFile('fpoe.txt', f3);
37   LoadFromFile('spoe.txt', f4);

39   (* file input cardinality *)
    WriteLn('== WS_ADT_Test ==');
41   WriteLn('- Input cardinality -');
    WriteLn(#9,'gruene: ', #9, Cardinality(f1));
43   WriteLn(#9,'oep: ', #9#9, Cardinality(f2));
    WriteLn(#9,'fpoe: ', #9#9, Cardinality(f3));
45   WriteLn(#9,'spoe: ', #9#9, Cardinality(f4));

47   (* file input cardinality with union *)
    WriteLn(#13#10,'- Input Union cardinality -');
49   WriteLn(#9,'spoe + oep: ', #9, Cardinality(Union(f4,f2)));
    WriteLn(#9,'gruene + oep: ', #9, Cardinality(Union(f1,f2)));
51   WriteLn(#9,'oep + fpoe: ', #9, Cardinality(Union(f2,f3)));
    WriteLn(#9,'fpoe + spoe: ', #9, Cardinality(Union(f3,f4)));
53   WriteLn(#9,'spoe + gruene: ', #9, Cardinality(Union(f4,f1)));

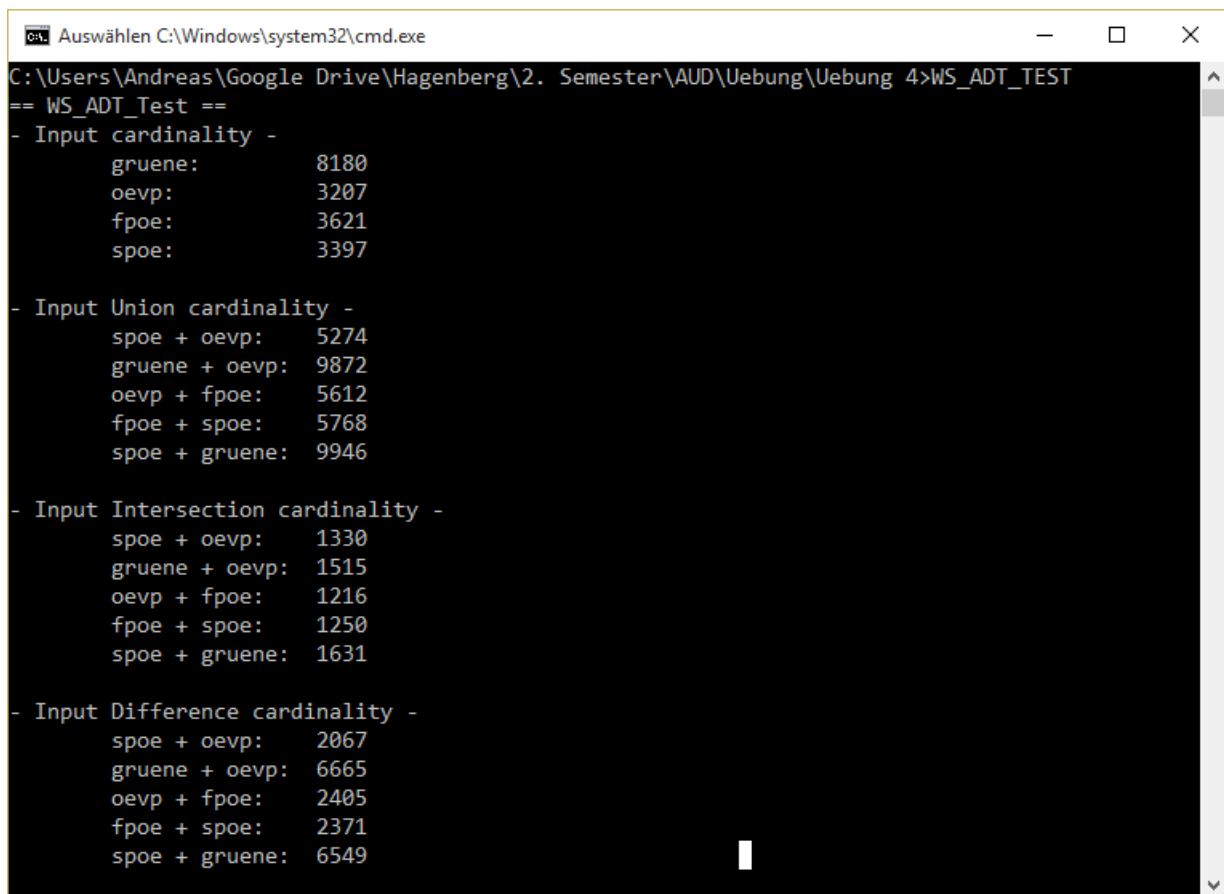
55   (* file input cardinality with intersection *)
    WriteLn(#13#10,'- Input Intersection cardinality -');
57   WriteLn(#9,'spoe + oep: ', #9, Cardinality(Intersection(f4,f2)));
    WriteLn(#9,'gruene + oep: ', #9, Cardinality(Intersection(f1,f2)));
59   WriteLn(#9,'oep + fpoe: ', #9, Cardinality(Intersection(f2,f3)));
    WriteLn(#9,'fpoe + spoe: ', #9, Cardinality(Intersection(f3,f4)));
61   WriteLn(#9,'spoe + gruene: ', #9, Cardinality(Intersection(f4,f1)));

63   (* file input cardinality with difference *)
    WriteLn(#13#10,'- Input Difference cardinality -');
65   WriteLn(#9,'spoe + oep: ', #9, Cardinality(Difference(f4,f2)));
    WriteLn(#9,'gruene + oep: ', #9, Cardinality(Difference(f1,f2)));
67   WriteLn(#9,'oep + fpoe: ', #9, Cardinality(Difference(f2,f3)));
    WriteLn(#9,'fpoe + spoe: ', #9, Cardinality(Difference(f3,f4)));
69   WriteLn(#9,'spoe + gruene: ', #9, Cardinality(Difference(f4,f1)));

71   DisposeWs(f1);
    DisposeWs(f2);
73   DisposeWs(f3);
    DisposeWs(f4);
75   END.

```

WS_ADT_TEST.pas



```
Auswählen C:\Windows\system32\cmd.exe
C:\Users\Andreas\Google Drive\Hagenberg\2. Semester\AUD\Uebung\Uebung 4>WS_ADT_TEST
== WS_ADT_Test ==
- Input cardinality -
    gruene:      8180
    oevp:        3207
    fpoe:        3621
    spoe:        3397

- Input Union cardinality -
    spoe + oevp:  5274
    gruene + oevp: 9872
    oevp + fpoe:  5612
    fpoe + spoe:  5768
    spoe + gruene: 9946

- Input Intersection cardinality -
    spoe + oevp:  1330
    gruene + oevp: 1515
    oevp + fpoe:  1216
    fpoe + spoe:  1250
    spoe + gruene: 1631

- Input Difference cardinality -
    spoe + oevp:  2067
    gruene + oevp: 6665
    oevp + fpoe:  2405
    fpoe + spoe:  2371
    spoe + gruene: 6549
```

Abbildung 3: WS ADT Unit Test

Bei diesen Testfällen enthält das WordSet von SPÖ und den GRÜNEN die meisten gemeinsamen Wörter, 1631. SPÖ und ÖVP haben “nur“ 1330 gemeinsame Wörter. Hier wird auch ersichtlich das die SPÖ und die GRÜNEN am zweiten Platz stehen bei den unterschiedlichen Wörtern, obwohl sie so viele gemeinsame Wörter haben.