Aufgabe 2 - ADT: Multiset

Lösungsidee:
Standardfunktionen eines BST einbauen für strings und helper Funktionen/Prozeduren verwenden, um so viel code Duplikation wie möglich zu vermeiden.

Zeitaufwand: ~1h 30min

Code:

```
unit Multiset;

interface

type
  PstrNode = ^StrNode;
  StrNode = record
    value: string;
    count: integer;
    left: PstrNode;
    right: PstrNode;
  end;
  StrMSet = record
    root: PstrNode;
  end;

procedure InitStrMSet(var ms: StrMSet);
procedure DisposeStrMSet(var ms: StrMSet);
procedure Insert(var ms: StrMSet; value: STRING);
procedure Remove(var ms: StrMSet; value: STRING);
function IsEmpty(ms: StrMSet): BOOLEAN;
function Contains(ms: StrMSet; value: STRING): BOOLEAN;
function Count(ms: StrMSet; value: STRING): INTEGER;
function Cardinality(ms: StrMSet): INTEGER;
function CountUnique(ms: StrMSet): INTEGER;
// helper procedure for easier debugging
procedure PrintTree(ms: StrMSet);

implementation

// helper functions
function NewStrNode(value: string): PstrNode;
var
  node: PstrNode;
begin
```

```
  New(node);
  node^.value := value;
  node^.count := 1;
  node^.left := nil;
  node^.right := nil;
  NewStrNode := node;
end;

function InsertStrNode(var node: PstrNode; value: string): PstrNode;
begin
  if node = nil then
  begin
    InsertStrNode := NewStrNode(value);
    Exit;
  end;

  if value < node^.value then
    node^.left := InsertStrNode(node^.left, value)
  else if value > node^.value then
    node^.right := InsertStrNode(node^.right, value)
  else
    Inc(node^.count);

  InsertStrNode := node;
end;

function RemoveStrNode(var node: PstrNode; value: string): PstrNode;
var
  successor: PstrNode;
begin
  if node = nil then
  begin
    RemoveStrNode := nil;
    Exit;
  end;

  if value < node^.value then
    node^.left := RemoveStrNode(node^.left, value)
  else if value > node^.value then
    node^.right := RemoveStrNode(node^.right, value)
  else if node^.count > 1 then
    Dec(node^.count)
  else if node^.left = nil then
  begin
    RemoveStrNode := node^.right;
    Dispose(node);
    Exit;
  end else if node^.right = nil then
```

```pascal
  begin
    RemoveStrNode := node^.left;
    Dispose(node);
    Exit;
  end else begin
    successor := node^.right;
    while successor^.left <> nil do
      successor := successor^.left;
    node^.value := successor^.value;
    node^.count := successor^.count;
    node^.right := RemoveStrNode(node^.right, successor^.value);
  end;

  RemoveStrNode := node;
end;

function FindStrNode(node: PStrNode; value: STRING): PStrNode;
begin
  if node = nil then
    FindStrNode := nil
  else if value < node^.value then
    FindStrNode := FindStrNode(node^.left, value)
  else if value > node^.value then
    FindStrNode := FindStrNode(node^.right, value)
  else
    FindStrNode := node;
end;

function CountNodeValues(node: PstrNode; uniqueOnly: Boolean): Integer;
begin
  if node = nil then
    CountNodeValues := 0
  else if uniqueOnly then
    CountNodeValues := 1 + CountNodeValues(node^.left, uniqueOnly) +
CountNodeValues(node^.right, uniqueOnly)
  else
    CountNodeValues := node^.count + CountNodeValues(node^.left, uniqueOnly) +
CountNodeValues(node^.right, uniqueOnly);
end;

procedure DisposeStrNode(node: PStrNode);
begin
  if node <> nil then
  begin
    DisposeStrNode(node^.left);
    DisposeStrNode(node^.right);
    Dispose(node);
  end;
```

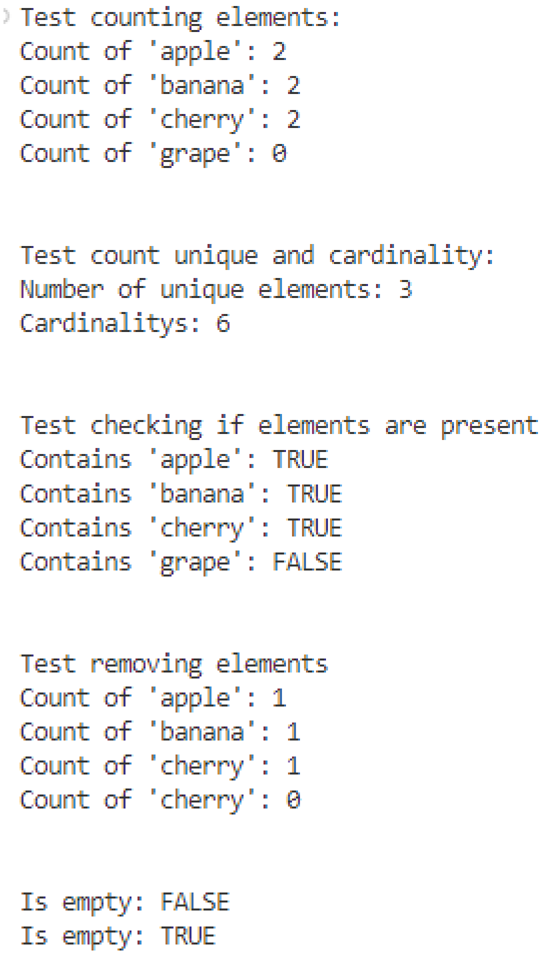
```pascal
end;

procedure PrintTreeNodes(root: PstrNode; level: integer);
var
  i: integer;
begin
  if root = nil then
    exit;

  PrintTreeNodes(root^.right, level + 1);

  for i := 1 to level do
    write('  ');

  writeln(root^.value, ':', root^.count);

  PrintTreeNodes(root^.left, level + 1);
end;

// helper functions end

procedure InitStrMSet(var ms: StrMSet);
begin
  ms.root := nil;
end;

procedure DisposeStrMSet(var ms: StrMSet);
begin
  if ms.root <> nil then
    DisposeStrNode(ms.root);
  ms.root := nil;
end;

procedure Insert(var ms: StrMSet; value: string);
begin
  ms.root := InsertStrNode(ms.root, value);
end;

procedure Remove(var ms: StrMSet; value: string);
begin
  ms.root := RemoveStrNode(ms.root, value);
end;

function IsEmpty(ms: StrMSet): Boolean;
begin
  IsEmpty := ms.root = nil;
end;
```

```
function Contains(ms: StrMSet; value: STRING): BOOLEAN;
begin
  Contains := FindStrNode(ms.root, value) <> nil;
end;

function Count(ms: StrMSet; value: STRING): INTEGER;
var
  node: PStrNode;
begin
  node := FindStrNode(ms.root, value);
  if node <> nil then
    Count := node^.count
  else
    Count := 0;
end;

function Cardinality(ms: StrMSet): Integer;
begin
  Cardinality := CountNodeValues(ms.root, False);
end;

function CountUnique(ms: StrMSet): Integer;
begin
  CountUnique := CountNodeValues(ms.root, True);
end;

procedure PrintTree(ms: StrMSet);
begin
  PrintTreeNodes(ms.root, 0);
end;

end.
```

Test Code:

```
program StrMSetTests;

uses Multiset;

var
  ms: StrMSet;
begin
  // Initialize an empty multiset
  InitStrMSet(ms);

  // Test inserting elements
  Insert(ms, 'apple');
  Insert(ms, 'banana');
  Insert(ms, 'apple');
  Insert(ms, 'cherry');
  Insert(ms, 'banana');
  Insert(ms, 'cherry');

  // Test counting elements
  WriteLn('Test counting elements:');
  WriteLn('Count of ''apple'': ', Count(ms, 'apple')); // should print 2
  WriteLn('Count of ''banana'': ', Count(ms, 'banana')); // should print 2
  WriteLn('Count of ''cherry'': ', Count(ms, 'cherry')); // should print 2
  WriteLn('Count of ''grape'': ', Count(ms, 'grape')); // should print 0
  WriteLn;
  WriteLn;

  // Test count unique and cardinality
  WriteLn('Test count unique and cardinality:');
  WriteLn('Number of unique elements: ', CountUnique(ms)); // should print 3
  WriteLn('Cardinalitys: ', Cardinality(ms)); // should print 6
  WriteLn;
  WriteLn;

  // Test checking if elements are present
  WriteLn('Test checking if elements are present');
  WriteLn('Contains ''apple'': ', Contains(ms, 'apple')); // should print True
  WriteLn('Contains ''banana'': ', Contains(ms, 'banana')); // should print
True
  WriteLn('Contains ''cherry'': ', Contains(ms, 'cherry')); // should print
True
  WriteLn('Contains ''grape'': ', Contains(ms, 'grape')); // should print
False
  WriteLn;
  WriteLn;

  // Test removing elements
  WriteLn('Test removing elements');
  Remove(ms, 'apple');
  WriteLn('Count of ''apple'': ', Count(ms, 'apple')); // should print 1
  Remove(ms, 'banana');
  WriteLn('Count of ''banana'': ', Count(ms, 'banana')); // should print 1
  Remove(ms, 'cherry');
  WriteLn('Count of ''cherry'': ', Count(ms, 'cherry')); // should print 1
  Remove(ms, 'cherry');
  WriteLn('Count of ''cherry'': ', Count(ms, 'cherry')); // should print 0
  WriteLn;
  WriteLn;
  // Test checking if multiset is empty
  WriteLn('Is empty: ', IsEmpty(ms)); // should print False

  // Test disposing of multiset
  DisposeStrMSet(ms);
  WriteLn('Is empty: ', IsEmpty(ms)); // should print True
end.
```

Test Ausgabe:



```
Test counting elements:
Count of 'apple': 2
Count of 'banana': 2
Count of 'cherry': 2
Count of 'grape': 0


Test count unique and cardinality:
Number of unique elements: 3
Cardinalitys: 6


Test checking if elements are present
Contains 'apple': TRUE
Contains 'banana': TRUE
Contains 'cherry': TRUE
Contains 'grape': FALSE


Test removing elements
Count of 'apple': 1
Count of 'banana': 1
Count of 'cherry': 1
Count of 'cherry': 0


Is empty: FALSE
Is empty: TRUE
```