

```
[ > restart:
  with(LinearAlgebra):
```

## – Entfernen von Duplikaten in Listen

Die im folgende verwendete Strategie ist, eine Liste der "Unikate" neu aufzubauen.

Dazu werden der Reihe nach alle Elemente der Ausgangsliste in eine anfangs leere Liste neu eingefügt.

Vor dem Einfügen wird aber jeweils geprüft, ob das Element bereits enthalten ist!

Das Enthalten-Sein überprüfen wir durch eine Hilfsfunktion `istElem`:

```
> istElem := proc(e, L::list)
  local i;
  for i in L do
    if i=e then return true; end if;
  end do;
  # Element wurde nicht gefunden
  return false;
end proc;
```

```
istElem := proc(e, L::list)
```

```
local i;
```

```
  for i in L do if i = e then return true end if end do; return false
```

```
end proc
```

Dann lautet die fertige Funktion `unique` wie folgt:

```
> unique := proc(L::list)
  local unique, elem;
  # Beginne mit einer leeren Liste unique:
  unique := [];
  # Schleife über alle Element der Ausgangsliste L:
  for elem in L do
    # Falls elem noch nicht in unique vorhanden, dann einfügen
    if istElem(elem, unique) = false
    then unique := [op(unique), elem];
    end if;
  end do;
  # Ergebnis steht in unique:
  return unique;
end proc;
```

```
unique := proc(L::list)
```

```
local unique, elem;
```

```
  unique := [ ];
```

```
  for elem in L do if istElem(elem, unique) = false then unique := [op(unique), elem] end if
```

```
  end do;
```

```
  return unique
```

```
end proc
```

Beispiel:

```

> unique( [1,4,7,3,5,7,9,5,4,2,3,5,8,4,2,6,4,7,8,2,1]);
[1, 4, 7, 3, 5, 9, 2, 8, 6]

```

## Binäre Suchbäume

Die Definition der Suchbäume erfolgt wie in der Vorlesung

Ein paar Beispielbäume:

```

> LEER := 'LEER';
d := [1, [LEER, LEER]];
b := [2, [d, LEER]];
e := [5, [LEER, LEER]];
c := [4, [LEER, e]];
a := [3, [b, c]];

```

*LEER := LEER*

*d := [1, [LEER, LEER]]*

*b := [2, [[1, [LEER, LEER]], LEER]]*

*e := [5, [LEER, LEER]]*

*c := [4, [LEER, [5, [LEER, LEER]]]]*

*a := [3, [[2, [[1, [LEER, LEER]], LEER]], [4, [LEER, [5, [LEER, LEER]]]]]]*

Um den kleinsten Knoten zu finden, müssen wir solange zum jeweils linken (kleineren) Sohn hinunter gehen, bis keiner mehr vorhanden ist:

```

> kleinsterKnoten := proc(baum)
    if baum[2][1] = LEER
    then return baum[1]
    else return kleinsterKnoten(baum[2][1])
    end if;
end proc;

```

*kleinsterKnoten := proc(baum)*

**if** *baum[2][1] = LEER* **then return** *baum[1]*

**else return** *kleinsterKnoten(baum[2][1])*

**end if**

**end proc**

Beispiele:

```

> kleinsterKnoten(c);
kleinsterKnoten(a);

```

4

1

Analog der größte Knoten:

```

> groessterKnoten := proc(baum)
    if baum[2][2] = LEER
    then return baum[1]
    else return groessterKnoten(baum[2][2])
    end if;
end proc;

```

```

groessterKnoten := proc(baum)
  if baum[2][2] = LEER then return baum[1]
  else return groessterKnoten(baum[2][2])
  end if

```

**end proc**

Beispiele:

```

> groessterKnoten(c);
groessterKnoten(a);

```

5

5

Für die Überprüfung, ob ein Binärbaum ein Suchbaum ist, könnten wir beinahe direkt die Definition abschreiben

```

> istSuchbaum1 := proc(baum)
  # ein leerer Baum ist ein Suchbaum
  if baum = LEER
  then return true
  else return
    groessterKnoten(baum[2][1]) < baum[1]
    and kleinstenKnoten(baum[2][2]) > baum[1]
    and istSuchbaum1(baum[2][1])
    and istSuchbaum2(baum[2][2])
  end if;
end proc;

```

```

istSuchbaum1 := proc(baum)

```

```

  if baum = LEER then return true

```

```

  else return groessterKnoten(baum[2][1]) < baum[1] and

```

```

    baum[1] < kleinstenKnoten(baum[2][2]) and istSuchbaum1(baum[2][1]) and

```

```

    istSuchbaum2(baum[2][2])

```

```

  end if

```

**end proc**

Leider funktioniert das nicht, weil die Funktionen groessterKnoten und kleinstenKnoten keine leeren Bäume als Parameter bekommen dürfen. (Vorsicht, die folgende Berechnung bricht nicht ab)

```

> istSuchbaum1(a);
Warning, computation interrupted

```

Deswegen müssen wir die and-Verknüpfungen im obigen Beispiel aufteilen und jeweils erst prüfen ob der linke bzw. rechte Teilbaum leer ist:

```

> istSuchbaum := proc(baum)
  # ein leerer Baum ist ein Suchbaum
  if baum = LEER then return true end if;
  # falls nicht alle Elemente des linken Teilbaums kleiner
  sind, dann falsch:
  if baum[2][1] <> LEER then
    if groessterKnoten(baum[2][1]) > baum[1] then return false
  end if

```

```

    end if;
    # falls nicht alle Elemente des rechten Teilbaums größer sind
    dann falsch:
    if baum[2][2] <> LEER then
        if kleinsterKnoten(baum[2][2]) < baum[1] then return false
    end if
    end if;
    # Die größer/kleiner-Forderungen sind erfüllt.
    # Jetzt müssen noch beide Söhne Suchbäume sein:
    if istSuchbaum(baum[2][1]) and istSuchbaum(baum[2][2])
    then return true
    else return false
    end if;
end proc;

```

*istSuchbaum* := **proc**(baum)

**if** baum = LEER **then return true end if;**

**if** baum[2][1] ≠ LEER **then**

**if** baum[1] < groessterKnoten(baum[2][1]) **then return false end if**

**end if;**

**if** baum[2][2] ≠ LEER **then**

**if** kleinsterKnoten(baum[2][2]) < baum[1] **then return false end if**

**end if;**

**if** istSuchbaum(baum[2][1]) **and** istSuchbaum(baum[2][2]) **then return true**

**else return false**

**end if**

**end proc**

Beispiele:

```

> istSuchbaum(a);
istSuchbaum([6,[a,b]]);
istSuchbaum([6,[a,LEER]]);
istSuchbaum([6,[LEER,a]]);

```

*true*

*false*

*true*

*false*

[ >

## – Matrizen und Skalarprodukt

Die Summenformeln lassen sich direkt in entsprechende for-Schleifen umwandeln:

```

[ > skalar := proc( v::Vector, A::Matrix, w::Vector, n::posint)
    local i,j, sumi, sumj;

```

```

    sumi := 0;
    for i from 1 to n do
        sumj := 0;
        for j from 1 to n do
            sumj := sumj + A[i,j]*w[j];
        end do;
        sumi := sumi + v[i]*sumj;
    end do;
    return sumi;
end proc;
skalar := proc(v::Vector, A::Matrix, w::Vector, n::posint)
local i, j, sumi, sumj;
    sumi := 0;
    for i to n do
        sumj := 0; for j to n do sumj := sumj + A[i, j]*w[j] end do; sumi := sumi + v[i]*sumj
    end do;
    return sumi
end proc
[ >

```

## – Matrizen und Eigenwerte

Die beiden Vektoren  $Av$  und  $\lambda v$  werden wir elementweise (also zeilenweise) vergleichen. Für die Berechnung eines einzelnen Elements des Vektors  $Av$  definieren wir eine Hilfsfunktion

```

[ > zeilenprodukt := proc(A::Matrix, v::Vector, zeile::posint,
    n::posint)
    local j, sum;
    sum := 0;
    for j from 1 to n do
        sum := sum + A[zeile, j]*v[j];
    end do;
    return sum;
end proc;
zeilenprodukt := proc(A::Matrix, v::Vector, zeile::posint, n::posint)
local j, sum;
    sum := 0; for j to n do sum := sum + A[zeile, j]*v[j] end do; return sum
end proc
[ > istEigenwert := proc( A::Matrix, v::Vector, lambda::numeric,
    n::posint)
    local i;
    # in jeder Zeile Zeilenprodukt mit lambda*v vergleichen
    for i from 1 to n do
        # Bei Nichtübereinstimmung ist sofort das ganze Ergebnis
        falsch:
        if zeilenprodukt(A, v, i, n) <> lambda*v[i]

```

```
        then return false;
        end if;
    end do;
    # Keine Zeile war falsch, also ist lambda Eigenwert
    return true;
end proc;
istEigenwert := proc(A::Matrix, v::Vector, λ::numeric, n::posint)
local i;
    for i to n do if zeilenprodukt(A, v, i, n) ≠ λ*v[i] then return false end if end do;
    return true
end proc
[ >
```