ADE/PRG v1.1

ADE/PRG

Jan Caspar, Aktualisiert 24. März 2017

Die Snippets wurden alle einmal kompiliert, ausgeführt und auf Korrektheit geprüft. Trotzdem kanns natürlich sein, dass irgendwas nicht ganz passt =P.

Die vom Georg gestohlenen hab ich nicht extra kompiliert und ausgeführt, ich geh mal davon aus er hat das gemacht. MIT, https://github.com/eisenwinter/fh-hgb-stuff

Teil I

Allgemein

Datentypen

elementare Typen

Primitiva: INTEGER, REAL, CHAR, WORD, BOOLEAN, Enum etc. etc.

benutzerdefinierte Typen

Bereichstypen (Ranges)

Beispiele [0..100], [Sa..So]

Aufzählungstypen (Enumerationen)

Beispiele Ampel = (rot,gelb,gruen); Tag = (Mo,Di,Mi,Do,Fr,Sa,So);

Strukturierte Datentypen

- sind auf anderen, einfacheren Datentypen aufgebaut.
- ermöglichen Aggregation von Einzelelementen

Beispiele: Set, Compound, Record, Array

Strutkurierte Datentypen gliedern sich in:

Statische Datenstrukturen

Beispiele Array, Verbund

Dynamische Datenstrukturen

Zur Definition von Datenstrukturen, bei denen zur Laufzeit nicht nur die zugeordneten Werte, sondern auch der Aufbau und die Größe variabel sein sollen, braucht man dynamische Strukturen

Beispiele: Listen, Bäume oder Graphen

Rekursion

Zu beachten ist einerseits das bei einem linear rekursiven Algorithmus schnell eine hohe Stacktiefe erreicht wird, was die Gefahr eines Stackoverflows birgt. Sollte er nicht linear rekursiv sein, so wird der Algorithmus langsam sein, allerdings dadurch nur sehr schwer eine hohe Stacktiefe erreichen.

linear rekursiv ein Algorithmus ist linear rekursiv, wenn pro rekursiven Zweig nur ein rekursiver Aufruf ist.

endrekursiv ein Algorithmus ist endrekursiv wenn nach dem rekursiven Aufruf keine Logik mehr kommt

Entrekursivieren

Ein Algorithmus lässt sich nur dann leicht in eine Iterative Lösung transformieren, wenn er endrekursiv und linear rekursiv ist. Ansonsten wird ein Hilfsstack benötigt.

Teil II

Algorithmen entwerfen

Vorgehensmodell

1. Klarheit über Aufgabenstellung verschaffen - Was ist gegeben, was ist gesucht?, Neben-, Ausnahme-, Umgebungsbedingungen, Beispiel-Szenarien)
2. Über Entwurfsstrategie entscheiden - z.B. schrittweise Verfeinerung
3. Lösungsidee entwickeln und dokumentieren - z.B. in stilisierter Prosa
4. Transformation der Lösungsidee in Algorithmus - (z.B. in Algorithmenbeschreibungssprache/Pseudocode)
5. Korrektheits- und Qualitätscheck der algorithmischen Lösung -(Korrespondenz zur Lösungsidee, Schnittstelle, Datenobjekte, Ablaufstruktur; Strukturqualität, Eleganz und Verständlichkeit)
6. Optimierung - (Ratschlag: Tu´s nicht – Erst wenn tatsächlich erforderlich Ggf. Re-Design überlegen Eleganz der Lösung)
7. Sicherstellung, dass alle Entwurfsentscheidungen dokumentiert sind
8. Transformation in ein ausführbares Programm(system) - (Beachtung programmiersprachenspezifischer Abweichungen vom Entwurf)
9. Systematischer Test - (statischer und dynamischer Test, Use Cases)

stepwise refinement

Zerlege eine Aufgabe in Teilaufgaben. Betrachte jede Teilaufgabe möglichst losgelöst von den anderen Teilaufgaben; zerlege sie weiter in Teilaufgaben, bis diese so einfach geworden sind, dass man dafür einen Algorithmus angeben kann.

Komplexität

Begriff und Abgrenzung

- Den Begriff Komplexität können wir also mit „Aufwand“ in Bezug setzen
- Laufzeitkomplexität (Zeitaufwand)
  - Wie lange braucht ein Algorithmus, um ein Ergebnis zu liefern?
  - Von welchen Parametern hängt die Laufzeit ab?
  - Wie ändert sich die Laufzeit wenn sich die „Problemgröße“ ändert?
- Speicherkomplexität (Speicheraufwand)
  - Wie viel Speicher braucht ein Algorithmus?
  - Von welchen Parametern hängt der Speicherbedarf ab?
- Strukturkomplexität (Test-/Verständnisaufwand)
  - Wie viele Verzweigungen sind in einem Algorithmus enthalten?

Laufzeitkomplexität

Die Laufzeitkomplexität ist eine Funktion einer Problemgröße n die den „Zeitaufwand“ zur Lösung der Aufgabe beschreibt

Typische Problemgrößen:

- Länge eines Texts oder Felds
- Anzahl der Knoten einer Liste oder eines Baums
- Größe einer Matrix
- Grad eines Polynoms

Grobanalyse (Laufzeitabschätzung)

- Ermittlung der Anzahl erforderlicher Schleifendurchläufe oder Prozeduraufrufen
- Details wie einzelne Anweisungen oder Ausdrucksauswertungen (sofern nicht essentiell) bleiben unberücksichtigt
- Analyse unabhängig vom verwendeten Prozessortyp und der verwendeten Programmiersprache

Vorgehensweise

- Bestimmen der für das Laufzeitverhalten wesentlichen Problemgröße
- Bestimmen der minimalen, maximalen und durchschnittlichen Anzahl der wesentlichen algorithmischen Schritte (z. B. Suchschritte, Schleifendurchläufe) in Abhängigkeit der Problemgröße

Feinanalyse (Laufzeitberechnung)

- Analyse jeder einzelnen Anweisung und jeder Ausdrucksauswertung
- Berechnung der Laufzeit bezogen auf einen bestimmten Prozessortyp und ggf. der eingesetzten Programmiersprache (Compiler)

Vorgehensweise

- Alle Anweisungen und Ausdrucksauswertungen werden berücksichtigt
  - Wie oft werden diese ausgeführt?
  - Wie lange dauert die Ausführung?
- Wir rechnen dabei nicht mit echten Ausführungszeiten, denn die sind prozessorabhängig, sondern mit Zeiteinheiten bezogen auf eine Referenzoperation (z. B. die Wertzuweisung = 1.0)

Ermitteln, wie oft jede Anweisung/jeder Ausdruck ausgeführt wird

x := 1 i := 1	1
while i ≤ n do	u+1
x := x * i	u
i := i + 1	u
end -- while	

Ausführungszeiten für jede Anweisung/jeden Ausdruck ermitteln

x := 1 i := 1	1	2.0	zwei Zuweisungen
while i ≤ n do	u+1	1.6	Vergleich
x := x * i	u	3.3	Zuweisung, Multiplikation
i := i + 1	u	1.8	Zuweisung, Addition
end -- while			

Gesamtausführungszeit ermitteln

$2.0 * 1 + 1.6(u + 1) + 3.3u + 1.8u = 3.6 + 6.7u$

Laufzeitmessung

Tatsächliches messen von Ausführungszeit, entweder über Tools oder selbst progammiert.

Teil III

Sortieren

Die Prozedur Swap steht für den Standard-Dreiecks-Tausch.

Bubble Sort

$O(n^2)$
Eine Familie besonders einfacher Sortierverfahren beruht auf der Lösungsidee, dass solange systematisch benachbarte Elemente miteinander verglichen und bei Bedarf vertauscht werden, bis der Datenbestand (in unserem Fall als Feld organisiert) sortiert ist. Da das Vertauschen benachbarter Elemente die zentrale Operation dieses Verfahren ist, nennt man es Austauschsortieren.

```
PROCEDURE BubbleSort(VAR arr : ARRAY OF INTEGER);
VAR
    i, j : Integer;
BEGIN
    FOR i := High(arr) DOWNTO Low(arr) DO
    BEGIN
        FOR j := Low(arr) TO i DO
        BEGIN
            If (arr[j−1] > arr[j]) Then
            BEGIN
                Swap(arr[j],arr[j−1]);
            END;
        END;
    END;
END;
```

Selection sort

$O(n^2)$
Zu Beginn wird das „kleinste“ Element aus dem zu sortierenden Feld ermittelt und mit dem ersten Element vertauscht (Swap). Danach wird mit dem Rest des Feldes wiederum so verfahren.

```
PROCEDURE SelectionSort(var arr : ARRAY OF INTEGER);
VAR
  i,j, minPos, minValue : INTEGER;
BEGIN
  FOR i := Low(arr) TO High(arr)−1 DO BEGIN
    minPos := i;
    minValue := arr[minPos];
    FOR j := i + 1 TO High(arr) DO
      IF arr[j] < minValue THEN
      BEGIN
        minPos := j;
        minValue := arr[minPos];
      END;
      Swap(arr[i],arr[minPos]);
  END;
END;
```

Combsort

$O(n^{1.3})$ Verbesserter Bubble Sort

```
PROCEDURE CombSort(VAR arr : ARRAY OF INTEGER);
VAR
  noSwaps : BOOLEAN;
  i,gap : LONGINT;
BEGIN
  gap := High(arr) − Low(arr) + 1;
  REPEAT
    gap := (gap * 10) DIV 13;
    IF gap = 0 THEN
      gap := 1;
    noSwaps := TRUE;
    FOR i := Low(arr) TO High(arr) − gap DO
      IF arr[i + gap] < arr[i] THEN BEGIN
        Swap(arr[i],arr[i+gap]);
        noSwaps := FALSE;
      END;
  UNTIL noSwaps AND (gap = 1);
END;
```

Insertion sort

$O(n^2)$
Zu Beginn betrachtet man das erste Element des Felds als sortierten Bereich und man sortiert das zweite Element, je nach seinem Schlüsselwert, vor oder hinter dem ersten ein. Der sortierte Bereich wird damit um ein Element vergrößert. Mit dem nächsten Element verfährt man ebenso: man fügt es im sortierten Bereich an der richtigen Stelle ein.

```
PROCEDURE InsertionSort(var arr : ARRAY OF INTEGER);
VAR
  i, j, h : LONGINT;
BEGIN
  FOR i:= Low(arr) TO High(arr) − 1 DO BEGIN
    h := arr[i+1];
    j := i;
    WHILE (j >= Low(arr)) AND (h < arr[j]) DO
    BEGIN
      arr[j+1] := arr[j];
      j := j − 1;
    END;
    arr[j+1]:= h;
  END;
END;
```

Shellsort

$O(n^{\frac{6}{5}})$ Verbesserter Insertion Sort.
Analysen haben gezeigt, dass auch für „fast sortierte“ Felder das Einfügesortieren eine annähernd lineare Laufzeitkomplexität aufweist (also günstig bleibt). Auf Basis ähnlicher Überlegungen hat Shell seinen Algorithmus so konstruiert, dass dieser in mehreren Schritten eine so gute „Vorsortierung“ des Felds herstellt, dass eine abschließende „Endsortierung“ mittels Einfügesortieren mit nur linear ansteigendem Aufwand möglich ist.

```
PROCEDURE ShellSort(VAR arr : ARRAY OF INTEGER);
VAR
  n,m,i,j,h: LONGINT;
BEGIN
  n := High(arr) − Low(arr) + 1;
  m := n DIV 2;
  WHILE m > 0 DO BEGIN
    FOR i := Low(arr) TO High(arr) − m DO
    BEGIN
      h := arr[i+m];
      j := i;
      WHILE (j >= Low(arr)) AND (h < arr[j]) DO
      BEGIN
        arr[j+m] := arr[j];
        j := j − m;
      END;
      arr[j+m]:= h;
    END;
    m := m DIV 2;
  END;
END;
```

Mergesort (Top-Down)

$O(n * log(n))$ Adaptiert von Java Implementierung von Sedgewick. Achtung! Bei offenem Array startet Index bei 0!
Es wird zunächst eine Teilungsposition ermittelt und dann wird mit rekursiven Aufrufen, jeweils eine Hälfte des Felds sortiert. Die eigentliche Sortierarbeit erfolgt dann folgendermaßen: die beiden sortierten Hälften des Felds werden (unter Zuhilfenahme des Hilfsalgorithmus Merge) zu einem sortierten Gesamtfeld zusammengemischt.

```
PROGRAM MergeSort;

PROCEDURE Merge(VAR a : ARRAY OF INTEGER; aux : Array OF INTEGER; lo,mid,hi : INTEGER);
VAR
    i,j,k : INTEGER;
BEGIN
    (* Merge a[lo..mid] with a[mid+1..hi]. *)
    i := lo;
    j := mid + 1;
    FOR k := lo TO hi DO (* Copy a[lo..hi] to aux[lo..hi] *)
        aux[k] := a[k];
    FOR k := lo TO hi DO (* Merge back to a[lo..hi]*)
    BEGIN
        IF i > mid THEN
        BEGIN
            a[k] := aux[j];
            Inc(j);
        END
        ELSE IF j > hi THEN
        BEGIN
            a[k] := aux[i];
            Inc(i);
        END
        ELSE IF aux[j] < aux[i] THEN
        BEGIN
            a[k] := aux[j];
            Inc(j);
        END
        ELSE
        BEGIN
            a[k] := aux[i];
            Inc(i);
        END;
    END;
END;

PROCEDURE MergeSort(VAR a : Array OF INTEGER; lo, hi : INTEGER);
VAR
    mid : INTEGER;
BEGIN
    IF lo < hi THEN
    BEGIN
        mid := lo + (hi − lo) DIV 2;
        MergeSort(a,lo,mid); (* sort left half *)
        MergeSort(a,mid+1,hi); (* sort right half *)
        Merge(a,a,lo,mid,hi); (* merge results *)
    END;
END;
```

Quicksort

$O(n * log(n))$ Ist zwar rekursiv hat aber eine bessere Laufzeit als alle anderen angeführten. Beginnt mit dem Wert genau in der Mitte und sortiert nach Links und Rechts „Haufen“ bzw „Stacks“, diese werden dann wieder in der Mitte geteilt und nach links und rechts sortiert. „divide et impera“ bzw. teile und herrsche Prinzip.

```
1 PROCEDURE QuickSort(VAR arr : ARRAY OF INTEGER; lo, hi : INTEGER);
2 VAR
3   i,j : LONGINT;
4   m : LONGINT;
5 BEGIN
6   i := lo;
7   j := hi;
8   m := arr[(i+j) DIV 2];
9   REPEAT
10    WHILE arr[i] < m DO Inc(i);
11    WHILE m < arr[j] DO Dec(j);
12    IF i <= j THEN
13    BEGIN
14      IF i <> j THEN (* optionales if (optimierung)*)
15        Swap(arr[i],arr[j]);
16      Inc(i);
17      Dec(j);
18    END;
19  UNTIL i > j;
20  IF lo < j THEN
21    QuickSort(arr,lo,j);
22  IF i < hi THEN
23    QuickSort(arr,i,hi);
24 END;
```

Weitere Sortierverfahren

IndirectSort

Wir haben bei der Komplexitätsanalyse der Sortierverfahren darauf hingewiesen, dass neben der Anzahl der Schlüsselvergleiche auch die Anzahl der Zuweisung von Datenobjekten in der Regel eine nicht zu vernachlässigende Auswirkung auf das Laufzeitverhalten hat. Sind die zu sortierenden Datenobjekte sehr groß, ist es zweckmäßig, den Aufwand für das Verschieben der Datenobjekte zu minimieren. Das kann man durch indirektes Sortieren erreichen. Dabei wird nicht der Datenbestand selbst sortiert, sondern ein Feld von Zeigern, die auf die entsprechenden Datenobjekte verweisen.

BucketSort

Der BucketSort (das sogenannte Fächersortieren) Unter gewissen Voraussetzungen (Einschränkungen) ist das Sortieren sogar in linearer Zeit, also mit einer asymptotischen Laufzeitkomplexität O(n) möglich. Wenn z. B. die Schlüsselwerte aus einem relativ kleinen numerischen Bereich 1:max stammen, kann man ein Hilfsfeld h mit max Elementen verwenden, in dem verkettete Listen aus Datenobjekten so verankert werden, dass jedes Datenobjekt unter Heranziehung seines Schlüsselwerts x als Index im Feld h in die entsprechende Liste h[x] eingefügt wird. Ein abschließender Durchlauf durch das Feld h und durch die darin verankerten Listen ermöglicht es, die Datenobjekte in eine sortierte Reihenfolge zu bringen. Das Feld h kann als „Schrank mit Fächern“ aufgefasst werden, in welche die entsprechenden Datenobjekte einsortiert werden. Deshalb wird dieses Verfahren auch als Fächersortieren (BucketSort) bezeichnet.

Stabilität von Sortierverfahren

Ein Sortierverfahren wird als stabil (stable) bezeichnet, wenn die relative Reihenfolge von Datenobjekten mit gleichem Schlüsselwert durch den Sortiervorgang unverändert bleibt. Stabilität ist eine Eigenschaft, die nur wenige Sortierverfahren aufweisen, die aber für bestimmte Anwendungen essentiell ist.

Beispiele für instabile Sortierverfahren Selectionsort, Shell-Sortieren, Combsort, Quicksort

Beispiele für stabile Sortierverfahren Insertionsort, Bubblesort, Mergesort

Teil IV

Zufallszahlen

Anwendungsgebiete

- Simulation natürlicher Vorgänge: um Phänomene mit zufälligem Verhalten
- darzustellen (z. B. kernphysikalische Prozesse, Verkehrsprobleme)
- Stichproben
- Monte-Carlo-Methoden
- Test von Algorithmen mit zufälligen Daten
- Programmierung von Spielen und Fragen der künstl. Intelligenz
- Kryptographie
- Animation

Lineare-Kongruenz-Methode (LKM)

Standardverfahren nach Derrick H. Lehmer (1949)

Algorithmus 1 $x_{n+1} = (a * x_n + c) mod m$

m ... Modul ⟶möglichst groß (Periodenlänge)

a ... Multiplikator ⟶2 ≤ a < m

c ... Inkrement ⟶ 0 ≤ c < m

x0 ... vorherige Zufallszahl ⟶ 0 ≤ x0 < m

xn+1 ... nächste Zufallszahl ⟶ 0 ≤ xn < m

Güte des Generators hängt von Wahl der Faktoren a, c, und m ab

Regeln zur Wahl von m, a, c (nach Knuth und Sedgewick)

1. m: möglichst groß, typisch 2k oder 2k-1 damit x mod m einfach berechnet werden kann
2. a: um rund eine Zehnerpotenz kleiner als m, also und von der Form a = ...g21 mit gerader Ziffer g (unregelmäßiges Bitmuster (z.B. 1101010111101)
3. c: c = 1, Startwert beliebig zwischen 0 und m-1

Gutes Beispiel: a = 3421, m = 216 und x_0 = 0

Methoden zur Verlängerung der Periodenlänge

Schieberegistermethode nach Tausworthe

- fülle ein Feld t[1:r] mit Zufallszahlen (z. B. mit IntRand)
- verknüpfe zwei Elemente (t[p] und t[q]) bitweise mit exklusivem Oder
- verschiebe den Feldinhalt um eine Stelle nach rechts
- verwende das Verknüpfungsergebnis als Zufallszahl und fülle t[1] damit

Gute Ergebnisse mit p = 31, q = r = 55 (nach Knuth)

Tabellenmethode nach MacLaren und Marsaglia

Initialisierung

- fülle Tabelle t[0:r-1] mit Zufallszahlen erzeugt mit Generator 1

Generierung einer Zufallszahl

- erzeuge mit Generator 2 eine Zufallszahl x aus dem Intervall 0:r-1
- verwende t[x] als gesuchte Zufallszahl
- überschreibe t[x] mit Zufallszahl erzeugt durch Generator 1

Implementierung

```
1 VAR
2   x : LONGINT;
3
4 CONST
5   M = 32768;
6
7 FUNCTION IntRand : INTEGER;
8 CONST
9   A = 3421;
10  K = 1;
11 BEGIN
12  x := (A * x + K) MOD M;
13  IntRand := x;
14 END;
15
16 FUNCTION RangeRand(n : INTEGER) : INTEGER;
17 VAR
18  k, ir : LONGINT;
19 BEGIN
20  k := (m DIV n) * n;
21  REPEAT
22    ir := IntRand;
23  UNTIL ir < k;
24  BetterRangeRand := ir MOD n;
25 END;
26
27
28 FUNCTION RealRand : REAL;
29 BEGIN
30  RealRand := IntRand / M;
31 END;
```

Prüfung der Güte von Zufallszahlengeneratoren

Häufigkeitstest:

Dieser Test dient dazu, die Auftrittshäufigkeiten der von einem Generator gelieferten Zufallszahlen zu ermitteln.

Serientest:

Dieser Test dient – im Unterschied zum Häufigkeitstest – dazu, die Auftrittshäufigkeit von Zahlenpaaren zu ermitteln.

Lückentest:

Der Lückentest untersucht, in welchen Abständen sich eine bestimmte Zufallszahl in der Folge wiederholt (für die lineare Kongruenzmethode ist das für alle Zahlen die Periodenlänge).

Läufetest:

Ein Lauf (run) ist ein Teilfolge von auf- oder absteigenden Elementen in einer größeren Folge. Der Läufetest untersucht, wie viele solcher auf oder absteigende Teilfolgen (also Läufe) einer bestimmten Länge in der Zahlenfolge enthalten sind.

Chi2-Test und Himmelstest:

Zwei weitere, für den praktischen Einsatz wichtige Tests von Zufallszahlenfolg

Teil V

Strukturen

Array

Struktur

```
TYPE
  IntArray = ARRAY[0..10] OF INTEGER;
```

Binäre Suche

Gibt Index zurück, -1 bei nicht gefunden.

```
FUNCTION BinarySearch(arr: IntArray; v : INTEGER) : INTEGER;
VAR
   x,l,r : INTEGER;
BEGIN
   l := Low(arr);
   r := High(arr);
   REPEAT
      x := (l+r) DIV 2;
      IF v < arr[x] THEN
         r:= x - 1
      ELSE
         l := x + 1
   UNTIL (v = arr[x]) OR (l > r);
   IF v = arr[x] THEN
      BinarySearch := x
   ELSE
      BinarySearch:= -1;
END;
```

Sequienteller Lauf (c) Georg Schinnerl

```
FUNCTION CountSeq(a: IntArray; n: INTEGER) : INTEGER;
VAR
  i,len,max: INTEGER;
BEGIN
  max := -1;
  len := 1;
  FOR i := 1 TO n - 1 DO BEGIN
    IF a[i] < a[i+1] THEN BEGIN
      INC(len);
      IF (len > max) THEN
         max := len;
    END ELSE
      len := 1;
  END;
  len := max;
  CountSeq := len;
END;
```

Shift (c) Georg Schinnerl

```
PROCEDURE ShiftArray(VAR a: IntArray; s: INTEGER);
VAR
  i, j: INTEGER;
  tmp: INTEGER;
BEGIN
  IF (s > 0) AND (s < n) AND (n > 1) THEN
  BEGIN
    FOR j := 1 TO s DO BEGIN
      tmp := a[n];
      FOR i := n DOWNTO 2 DO BEGIN
        a[i] := a[i-1];
      END;
      a[1] := tmp;
    END;
  END;
END;
```

Single Linked Linea List (SLL)

Dieses Beispiel geht von einem einfachen Integer-Wert als Value.

Struktur

```
TYPE
  Node = ^NodeRec; (* Pointer Node *)
  NodeRec = RECORD
    value : INTEGER; (* Value *)
    next: Node; (* next node *)
  END;
  List = Node;
```

Überlaufspattern

```
PROCEDURE DoSomethingWithList(l : List);
VAR
  n : Node;
BEGIN
  n := l;
  (* as long as there is a next node *)
  WHILE n <> NIL DO BEGIN
    (* do something *)
    n := n^.next;
  END;
END;
```

initialisieren der Liste

```
PROCEDURE InitList(VAR l: List);
BEGIN
  l := NIL;
END;
```

neuer Knoten

```
FUNCTION NewNode(value : INTEGER) : Node;
VAR
  n : Node;
BEGIN
  New(n);
  n^.value := value;
  n^.next := NIL;
  NewNode := n;
END;
```

Contains

```
FUNCTION Contains(l : List; value : INTEGER) : BOOLEAN;
VAR
  n : Node;
BEGIN
  n := l;
  (* as long as there is a node and we havent found the value *)
  WHILE (n <> NIL) AND (n^.value <> value) DO
    n := n^.next;
  Contains := n <> NIL;
END;
```

Count

```
FUNCTION Count(l : List; value : INTEGER) : INTEGER;
VAR
  c : INTEGER;
  n : Node;
BEGIN
  c := 0;
  n := l;
  WHILE (n <> NIL) DO
  BEGIN
   IF value = n^.value THEN
      c := c + 1;
    n := n^.next;
  END;
  Count := c;
END;
```

Append

```
PROCEDURE Append(VAR l: List; value : INTEGER);
VAR
  n : Node;
BEGIN
  IF l = NIL THEN
    l := NewNode(value)
  ELSE BEGIN
    n := l;
    WHILE n^.next <> NIL DO
      n := n^.next;
    n^.next := NewNode(value);
  END;
END;
```

Prepend

```
PROCEDURE Prepend(VAR l : List; value : INTEGER);
VAR
  n : Node;
BEGIN
  n := NewNode(value);
  n^.next := l;
  l := n;
END;
```

Insert After First

Anmerkung: der Wert wird hinten angehängt wenn er nicht gefunden wird, sollte er nur bei gefundenen Wert anhängen einfach den Block weglassen.

```
PROCEDURE InsertAfterFirst(VAR l : List; search : INTEGER; value : INTEGER);
VAR
  n,nxt,cr : Node;
BEGIN
  n := l;
  nxt := l;
  WHILE (n <> NIL) AND (n^.value <> search) DO
  BEGIN
    nxt := n;
    n := n^.next;
  END;
  IF n <> NIL THEN
  BEGIN
    nxt := n^.next;
    cr := NewNode(value);
    n^.next := cr;
    cr^.next := nxt;
  END
 (* comment for insertion only on exist *)
  ELSE IF nxt <> NIL THEN
  BEGIN
    cr := NewNode(value);
    nxt^.next := cr;
  END
  ELSE (* is empty list *)
    l := NewNode(value);
  (* end ins. without found *)
END;
```

Insert After Last

Reverse -> InsertAfterFirst -> Reverse

Merge

```
PROCEDURE Merge(VAR lo : List; hi : List);
VAR
    n : Node;
BEGIN
    IF (lo <> NIL) AND (hi <> NIL) THEN
    BEGIN
        n := lo;
        WHILE n^.next <> NIL DO
            n := n^.next;
        n^.next := hi;
    END
    ELSE IF (hi <> NIL) THEN
        lo := hi;
END;
```

Delete

Löscht ersten gefundenen

```
PROCEDURE Delete(VAR l : List; value : INTEGER);
VAR
    n,prev : Node;
BEGIN
  n := l;
  WHILE (n <> NIL) AND (n^.value <> value) DO
  BEGIN
      prev := n;
    n := n^.next;
  END;
  IF (n <> NIL) THEN
  BEGIN
   prev^.next := n^.next;
   Dispose(n);
  END;
END;
```

Distinct

Neue Liste ohne Duplikate.

```
FUNCTION Distinct(l : List) : List;
var
   distinctList : List;
   n : Node;
BEGIN
  n := l;
  distinctList := NIL;
  WHILE n <> NIL DO BEGIN
    IF NOT Contains(distinctList,n^.value) THEN
       Prepend(distinctList,n^.value);
    n := n^.next;
  END;
  Reverse(distinctList);
  Distinct := distinctList;
END;
```

Reverse

```
PROCEDURE Reverse(VAR l : List);
VAR
  invList : List;
  next : Node;
BEGIN
  IF (l <> NIL) AND (l^.next <> NIL) THEN
  BEGIN
    invList := l;
    l := l^.next;
    invList^.next := NIL;
    WHILE l <> NIL DO
   BEGIN
       next := l^.next;
       l^.next := invList;
       invList := l;
       l := next;
    END;
    l := invList;
  END;
END;
```

Clear

```
PROCEDURE ClearList(VAR l : List);
VAR
  n, next : Node;
BEGIN
  n := l;
  WHILE n <> NIL DO BEGIN
    next := n^.next;
    Dispose(n);
    n := next;
  END;
  l := NIL;
END;
```

Dispose

```
PROCEDURE DisposeList(l : List);
BEGIN
  ClearList(l);
END;
```

Selection Sort (c) Georg Schinnerl

```
PROCEDURE SelectionSort(VAR l: List);
VAR
  n, m, nMin: Node;
  min: INTEGER;
  tmp: INTEGER;
BEGIN
  IF l <> NIL THEN BEGIN
    n := l;
    WHILE (n^.next <> NIL) DO BEGIN
      nMin := n;
      min := nMin^.data;
      m := n^.next;
      WHILE (m <> NIL) DO BEGIN
        IF m^.data < min THEN BEGIN
          nMin := m;
          min := nMin^.data;
        END;
        m := m^.next;
      END;
      (* swap *)
      tmp := nMin^.data;
      nMin^.data := n^.data;
      n^.data := tmp;
      n := n^.next;
    END;
  END;
END;
```

Double Linked Cylcic Anchor List (DLCA)

Dieses Beispiel geht von einem einfachen Integer-Wert als Value.

Struktur

```
TYPE
  Node = ^NodeRec;
  NodeRec = RECORD
    value : INTEGER;
    prev, next: Node;
  END;
  List = Node;
```

Überlaufspattern

```
PROCEDURE DoSomethingWithList(l : List);
VAR
  n : Node;
BEGIN
  n := l^.next;
  WHILE n <> l DO BEGIN
    (* do something with n^.value *)
    n := n^.next;
  END;
END;
```

initialisieren der Liste

```
PROCEDURE InitList(VAR l: List);
BEGIN
  l := NewNode(o);
END;
```

neuer Knoten

```
FUNCTION NewNode(value : INTEGER) : Node;
VAR
  n : Node;
BEGIN
  New(n);
  n^.value := value;
  n^.next := n;
  n^.prev := n;
  NewNode := n;
END;
```

Contains

```
FUNCTION Contains(l : List; value : INTEGER) : BOOLEAN;
VAR
  n : Node;
BEGIN
  n := l^.next;
  WHILE (n <> l) AND (n^.value <> value) DO
    n := n^.next;
  Contains := n <> l;
END;
```

Append

```
PROCEDURE Append(l: List; value : INTEGER);
VAR
    n : Node;
BEGIN
  n := NewNode(value);
  n^.next := l;
  n^.prev := l^.prev;
  l^.prev^.next := n;
  l^.prev := n;
END;
```

Prepend

```
PROCEDURE Prepend(l : List; value : INTEGER);
VAR
  n : Node;
BEGIN
  n := NewNode(value);
  n^.prev := l;
  n^.next := l^.next;
  l^.next^.prev := n;
  l^.next := n;
END;
```

Clear

```
PROCEDURE ClearList(l : List);
VAR
  n, next : Node;
BEGIN
  n := l^.next;
  WHILE n <> l DO BEGIN
    next := n^.next;
    Dispose(n);
    n := next;
  END;
  l^.next := l;
END;
```

Dispose

```
PROCEDURE DisposeList(l : List);
BEGIN
  ClearList(l);
  Dispose(l);
END;
```

Binary Search Tree (BST)

Dieses Beispiel geht von einem einfachen Integer-Wert als Value.

Struktur

```
TYPE
  Node = ^NodeRec;
  NodeRec = RECORD
    value : INTEGER;
    left, right : Node;
  END;
  Tree = Node;
```

Überlaufspattern - In Order (aufsteigend sortiert)

```
PROCEDURE DoSomethingWithTree(t : Tree);
BEGIN
  IF t <> NIL THEN
  BEGIN
    DoSomethingWithTree(t^.left);
    (* do something with t^.value *)
    DoSomethingWithTree(t^.right);
  END;
END;
```

Überlaufspattern - In Order Reverse (absteigend sortiert)

```
PROCEDURE DoSomethingWithTree(t : Tree);
BEGIN
  IF t <> NIL THEN
  BEGIN
    DoSomethingWithTree(t^.right);
    (* do something with t^.value *)
   DoSomethingWithTree(t^.left);
  END;
END;
```

initialisieren des Baums

```
PROCEDURE InitTree(VAR t: TREE);
BEGIN
  t := NIL;
END;
```

neuer Knoten

```
FUNCTION NewNode(value : INTEGER) : Node;
VAR
  n : Node;
BEGIN
  New(n);
  n^.value := value;
  n^.next := n;
  n^.prev := n;
  NewNode := n;
END;
```

Contains

```
FUNCTION ContainsValue(t : Tree; value : INTEGER) : BOOLEAN;
BEGIN
  WHILE (t <> NIL) AND (value <> t^.value) DO
  BEGIN
    IF value < t^.value THEN
      t := t^.left
    ELSE
      t := t^.right;
  END;
  ContainsValue := t <> NIL;
END;
```

Add

```
PROCEDURE AddValue(var t: Tree; value : INTEGER);
VAR
  newNode,prev,n : Node;
BEGIN
  New(newNode);
  newNode^.value := value;
  newNode^.left := NIL;
  newNode^.right := NIL;

  IF t = NIL THEN
    t := newNode
  ELSE
  BEGIN
    n := t;
    WHILE n <> NIL DO
    BEGIN
      prev := n;
      IF value < t^.value THEN
        n := n^.left
     ELSE
        n := n^.right;
    END;
    IF value < prev^.value THEN
      prev^.left := newNode
    ELSE
      prev^.right := newNode;
  END;
END;
```

Clear

```
PROCEDURE ClearTree(VAR t : Tree);
BEGIN
  IF t <> NIL THEN
  BEGIN
   ClearTree(t^.left);
    ClearTree(t^.right);
    Dispose(t);
    t := NIL;
  END;
END;
```

Dispose

```
PROCEDURE DisposeTree(t : Tree);
BEGIN
  ClearTree(t);
END;
```

String from Tree (c) Georg Schinnerl

```
FUNCTION StringFromTree(t: Tree; key: INTEGER): STRING;
BEGIN
  IF t = NIL THEN
    StringFromTree := ''
  ELSE IF key = t^.key THEN
    (* effizienteste methode, wenn hier nur mehr der rechte Baum berücksichtigt wird *)
    StringFromTree := t^.c + StringFromTree(t^.right, key)
  ELSE IF key < t^.key THEN
    StringFromTree := StringFromTree(t^.left, key)
  ELSE StringFromTree := StringFromTree(t^.right, key);
END;
```

Inhaltsverzeichnis

I Allgemein 1

Datentypen 1
- elementare Typen 1
  - benutzerdefinierte Typen 1
- Strukturierte Datentypen 1
  - Statische Datenstrukturen 1
  - Dynamische Datenstrukturen 1

Rekursion 1
- Entrekursivieren 1

II Algorithmen entwerfen 1

Vorgehensmodell 1
- stepwise refinement 1

Komplexität 1
- Begriff und Abgrenzung 1
- Laufzeitkomplexität 1
- Grobanalyse (Laufzeitabschätzung) 1
  - Vorgehensweise 1
- Feinanalyse (Laufzeitberechnung) 1
  - Vorgehensweise 1

Laufzeitmessung 1

III Sortieren 2

Bubble Sort 2

Selection sort 2

Combsort 2

Insertion sort 2

Shellsort 2

Mergesort (Top-Down) 2

Quicksort 3

Weitere Sortierverfahren 3
- IndirectSort 3
- BucketSort 3

Stabilität von Sortierverfahren 3

IV Zufallszahlen 3

Anwendungsgebiete 3

Lineare-Kongruenz-Methode (LKM) 3
- Methoden zur Verlängerung der Periodenlänge 3
  - Schieberegistermethode nach Tausworthe 3
  - Tabellenmethode nach MacLaren und Marsaglia 3
- Implementierung 3

Prüfung der Güte von Zufallszahlengeneratoren 3

V Strukturen 4

Array 4
- Struktur 4
- Binäre Suche 4
- Sequienteller Lauf (c) Georg Schinnerl 4
- Shift (c) Georg Schinnerl 4

Single Linked Linea List (SLL) 4
- Struktur 4
- Überlaufspattern 4
- initialisieren der Liste 4
- neuer Knoten 4
- Contains 4
- Count 4
- Append 4
- Prepend 4
- Insert After First 5
- Insert After Last 5
- Merge 5
- Delete 5
- Distinct 5
- Reverse 5
- Clear 5
- Dispose 5
- Selection Sort (c) Georg Schinnerl 5

Double Linked Cylcic Anchor List (DLCA) 6
- Struktur 6
- Überlaufspattern 6
- initialisieren der Liste 6
- neuer Knoten 6
- Contains 6
- Append 6
- Prepend 6
- Clear 6
- Dispose 6

Binary Search Tree (BST) 7
- Struktur 7
- Überlaufspattern - In Order (aufsteigend sortiert) 7
- Überlaufspattern - In Order Reverse (absteigend sortiert) 7
- initialisieren des Baums 7
- neuer Knoten 7
- Contains 7
- Add 7
- Clear 7
- Dispose 7
- String from Tree (c) Georg Schinnerl 7