Allgemeine Informatik II
im SS 2005

Andreas F. Borchert
Abteilung Angewandte
Informationsverarbeitung

14. September 2005

http://www.mathematik.uni-ulm.de/sai/ss05/ai2/

Einführende Hinweise

Die Vorlesung Allgemeine Informatik II soll der Einführung und ersten Orientierung folgender Themen dienen:

- Rekursion, Produktionssysteme, einfaches Recursive-Descent-Parsing, Back-Tracking-Verfahren, Branch-And-Bound-Verfahren.
- Modularisierung, abstrakte Datentypen.
- Dynamische Datenstrukturen: Zeiger, lineare Listen, Bäume, Hash-Verfahren.
- Mengen, topologisches Sortieren, transitive Hüllen.

Literaturhinweise

Es gibt kein Buch, an das sich die Vorlesung in enger Form anlehnt. Als Begleitlektüre könnten folgende Werke interessant sein:

- Donald E. Knuth, “The Art of Computer Programming”, Bände 1 bis 3, Addison-Wesley.
  Diese Bände sind absolute Standardwerke, die vor einigen Jahren in einer überarbeiteten Fassung erschienen sind. Sie gehen u.a. auf Datenstrukturen, Zufallszahlen, Arithmetik, Sortieren und Suchen in großer Tiefe und beeindruckender Breite ein.

- Niklaus Wirth, “Algorithmen und Datenstrukturen”, Teubner-Verlag.
  Geht ein auf Datenstrukturen, Sortieren, rekursive Algorithmen und dynamische Informationsstrukturen mit vielen Beispielen in Modula-2.

- Robert Sedgewick, “Algorithms in Modula-3”, Addison-Wesley.
  Geht auf ein sehr breites Spektrum an Themen ein, die jeweils kurz vorgestellt werden und mit einem vollständigen Beispiel in Modula-3 versehen sind.

- Martin Reiser und Niklaus Wirth, “Programming in Oberon”, Addison-Wesley.
  Einführung in Oberon und objekt-orientierte Techniken mit Oberon.

Weiterführende Veranstaltungen

Diese Vorlesung kann im Rahmen der zur Verfügung stehenden Zeit nur eine Einführung geben. Folgende Lehrveranstaltungen eignen sich zur weiterführenden Vertiefung:

- Algorithmen I und II
- Parallele Algorithmen
- Komplexitätstheorie
- Objektorientierte Programmierung mit Java
- C++ mit Data-Mining-Anwendungen

Übungen und Klausuren

- Bitte melden Sie sich zur Vorlesung an unter `https://slc.mathematik.uni-ulm.de/`

- Die Übungen betreut Norbert Heidenbluth, zu erreichen unter `heidi@mathematik.uni-ulm.de`

- Die ersten Übungen und das erste Übungsblatt gibt es am Donnerstag, den 14. April. Zu diesem Termin werden auch die Tutoren vorgestellt.

- Für den Erwerb eines Scheins sind 50% der Übungspunkte und 50% Punkte bei der Klausur erforderlich.

- Zur Klausur sind nur diejenigen zugelassen, die genügend Übungspunkte erhalten haben.

- Die Klausur findet statt am Samstag, den 9. Juli 2005, in der Zeit von 10:00 bis 12:00 Uhr.

- Eine Nachklausur wird es an einem noch festzulegenden Termin Ende September oder Anfang Oktober geben.

Tutorien

- Übungspunkte werden in Tutorien vergeben.
- Bitte nehmen Sie darauf Rücksicht, daß ein Tutor etwa 20 Vorlesungsteilnehmer zu betreuen hat.
- Entsprechend sollten Sie gut vorbereitet zu einem Tutorium kommen. Sie können Hilfe in konkreten Fragen erwarten, jedoch keine Wiederholung des gesamten Stoffes.
- Die Tutoren sind berechtigt, ein Tutorium im Falle mangelnder Vorbereitung abzubrechen.
- Nach erfolgter Tutorzuteilung können Tutoren nur noch in begründeten Ausnahmefällen gewechselt werden.
- Beschwerden sind an den Übungsleiter zu richten.

Ethik

- Es ist Ihnen gestattet, sich mit anderen Kommilitonen auch außerhalb Ihrer Gruppe über das aktuelle Übungsblatt auszutauschen.

- Die Übernahme von Programmtexten von anderen ist dabei nur zulässig, wenn

  - der Autor damit einverstanden ist,

  - Kommentare in Ihrem Programmtext klarlegen, welche Teile von wem ursprünglich geschrieben wurden und

  - auch die übernommenen Teile vollständig verstanden sind, so daß sie im Tutorium erklärt werden können.

Auffrischungen

Wir wollen testweise ein spezielles Angebot denjenigen machen, die den Anschluß in Allgemeine Informatik verloren haben und mit sehr großen Schwierigkeiten kämpfen, selbst Programmtexte zu verfassen.

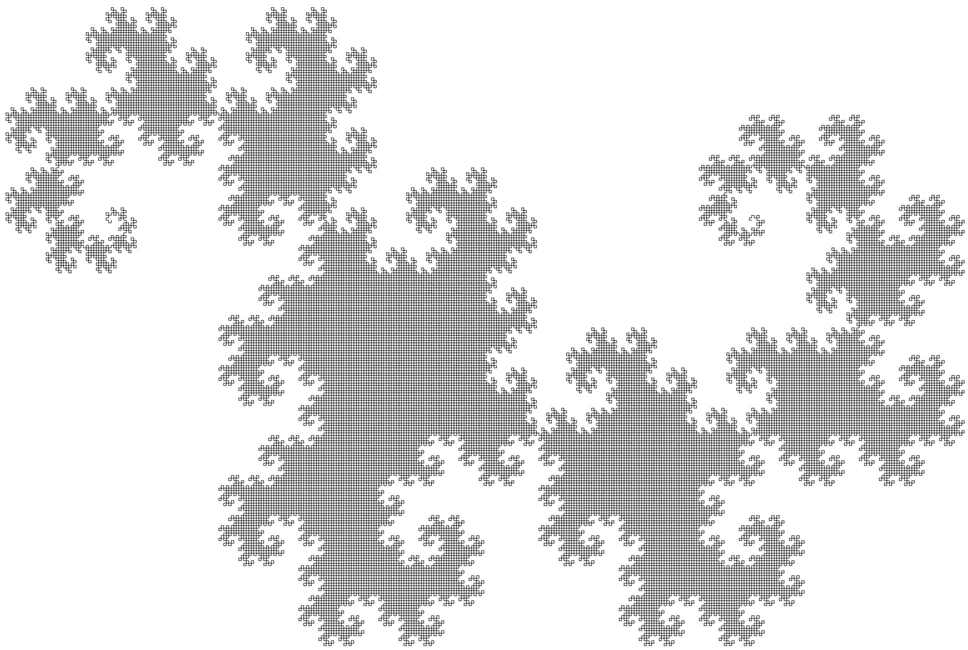
- Am Freitag-Vormittag treffen wir uns um 8 Uhr im Raum O27/123.
- Zu einer ausgewählten Thematik wird es eine kurze Wiederholung im Umfange von 30-45 Minuten geben im Seminarraum.
- Danach besteht die Gelegenheit, in den Poolraum O27/213 zu wandern und eine kleine überschaubare Übungsaufgabe zu lösen mit unmittelbarer Betreuung von Herrn Heidenbluth und mir.
- Am ersten Termin (15. April) werden wir uns mit eindimensionalen Arrays beschäftigen.

Sprechstunden

- Sie können mich jederzeit per E-Mail kontaktieren:
  `borchert@mathematik.uni-ulm.de`

- Sie sind eingeladen, auch bei mir persönlich vorbei zu schauen. Gute Zeiten sind Mittwoch, Donnerstag 10-12 und 14-18 Uhr. Mein Büro ist in der Helmholtzstraße 18, Zimmer E02 (Erdgeschoß, rechter Flügel, dritte Tür rechts).

Rekursion



- Viele Probleme, Modelle oder Phänomen haben eine sich selbst referenzierende Form, bei der die eigene Struktur immer wieder in unterschiedlichen Varianten enthalten ist.

- Wenn diese Strukturen in eine mathematische Definition, einen Algorithmus oder eine Datenstruktur übernommen werden, wird von Rekursion gesprochen.

- Rekursive Definitionen sind jedoch nur sinnvoll, wenn etwas immer duch einfachere Versionen seiner selbst definiert wird, wobei im Grenzfall ein Trivialfall gegeben ist, der keine Rekursion benötigt.

Rekursion in der Sprache

Rekursion hat mit Verschachteln zu tun und dies sind wir auch in unserer deutschen Sprache sehr gewohnt. Ein klassisches Beispiel hierfür ist Christian Morgensterns Vorrede zu seinen *Galgenliedern*:

> Es darf daher getrost, was auch von allen, deren Sinne, weil sie unter Sternen, die, wie der Dichter sagt, zu dörren, statt zu leuchten, geschaffen sind, geboren sind, vertrocknet sind, behauptet wird, enthauptet werden, daß hier einem sozumaßen und im Sinne der Zeit, dieselbe im Negativen als Hydra betrachtet, hydratherapeutischen Moment ersten Ranges, immer angesichts dessen, daß, wie oben, keine mit Rosenfingern den springenden Punkt ihrer schlechthin unvoreingenommenen Hoffnung auf eine, sagen wir, schwansinnige oder wesenzielle Erweiterung des natürlichen Stofffeldes zusamt mit der Freiheit des Individuums vor dem Gesetz ihrer Volksseele zu verraten den Mut, was sage ich, die Verruchtheit haben wird, einem Moment, wie ihm Handel, Wandel, Kunst und Wissenschaft allüberall dieselbe Erscheinung, dieselbe Tendenz den Arm bietet, und welches bei allem, ja vielleicht eben trotz allem, als ein mehr oder minder undulationsfähiger Ausdruck einer ganz bestimmten und im weitesten Verfolge excösen Weltauffasseraumwortkindundkunstanschauudng kaum mehr zu unterschlagen versucht werden zu wollen vermag – gegenübergestanden und beigewohnt werden zu dürfen gelten lassen zu müssen sein möchte.

Rekursive Definitionen

- Viele Mengen können am einfachsten rekursiv definiert werden. Dies gilt insbesondere dann, wenn eine Menge unendlich viele Elemente besitzt.

- Die Menge der natürlichen Zahlen kann wie folgt definiert werden:

  1. 1 ist eine natürliche Zahl.

  2. Wenn x eine natürliche Zahl ist, dann ist $x + 1$ ebenfalls eine natürliche Zahl.

- Ist 3 eine natürliche Zahl? Ja, weil

  - 1 ist eine natürliche Zahl (Regel 1),

  - 2 ist eine natürliche Zahl, weil $2 = 1 + 1$ (Regel 2) und

  - 3 ist eine natürliche Zahl, weil $3 = 2 + 1$ (Regel 2).

- Beginnend mit 1 (Regel 1) sind wir in der Lage, alle anderen natürlichen Zahlen durch die wiederholte Anwendung der zweiten Regel zu erzeugen.

Rekursive Definitionen

- Rekursive Definitionen können ebenso für Folgen oder für Funktionen verwendet werden.

- Die Fakultät $F(n) = n!$ ist für $n > 0$ definiert als

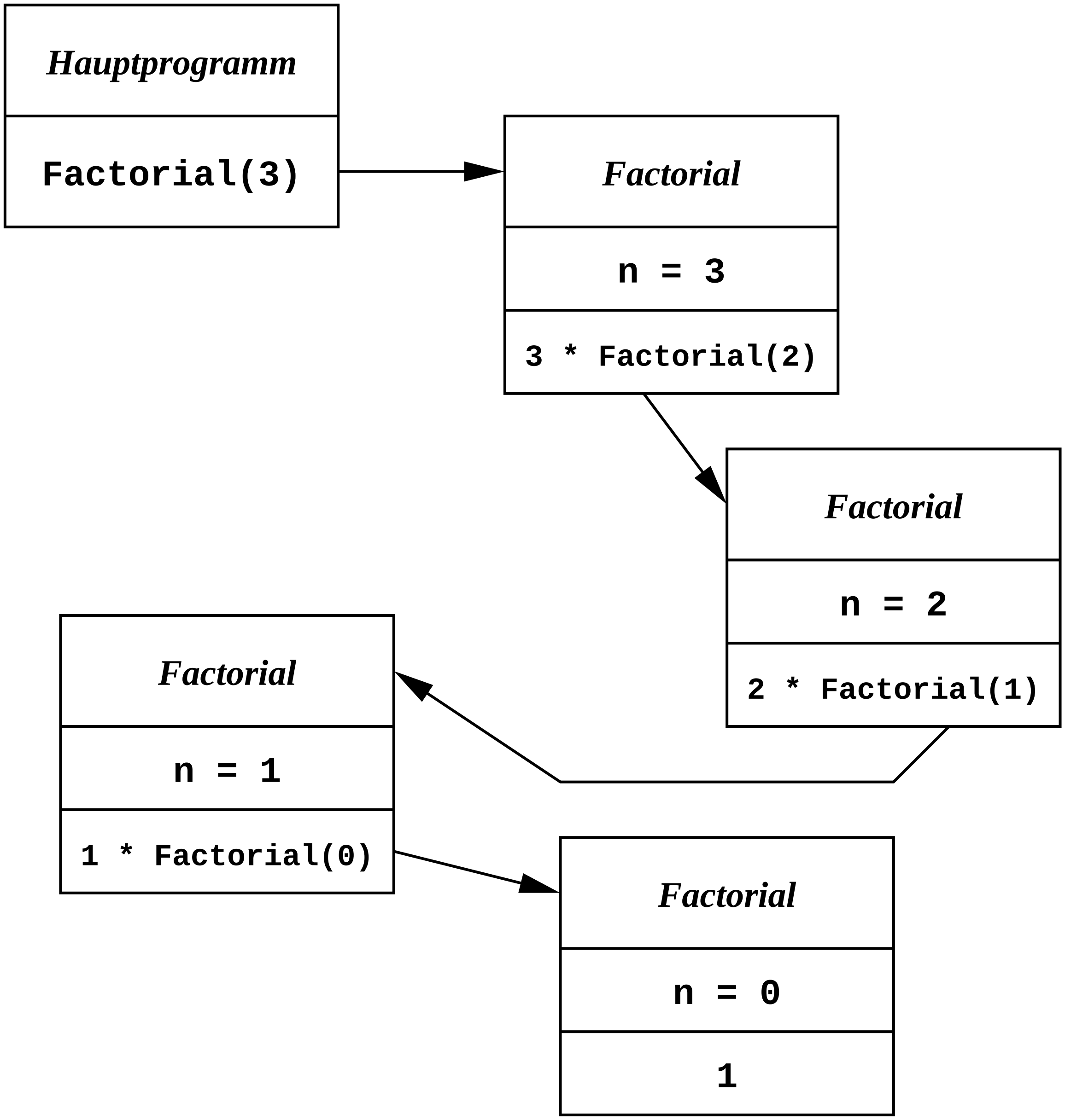
$$
\begin{aligned}
F(0) &= 1 \\
F(n) &= n \cdot F(n-1) \quad \text{für} \quad n > 0
\end{aligned}
$$

- In Oberon können Prozeduren sich selbst aufrufen. Damit ist es relativ leicht möglich, eine rekursive Definition mehr oder weniger direkt zu übernehmen:

Factorial.om

```
PROCEDURE Factorial(n: INTEGER) : INTEGER;
BEGIN
   IF n > 0 THEN
      RETURN n * Factorial(n - 1)
   ELSE
      RETURN 1
   END;
END Factorial;
```

Inkarnationen einer Prozedur



- Durch Rekursion ist es möglich, daß mehrere Inkarnationen einer Prozedur gleichzeitig existieren.

- Jede dieser Inkarnationen hat ihre eigenen Variablen und Parameter (in diesem Beispiel der Parameter n).

Endlichkeit der Rekursion

```
PROCEDURE Recursive(....);
BEGIN
   IF (* einfacher Fall *) THEN
      (* nicht-rekursiv *)
   ELSE
      (* ... *)
      Recursive(...);
      (* ... *)
   END;
END Recursive;
```

- Da nur begrenzte Rechenzeit zur Verfügung steht, muß auf die Endlichkeit der Rekursion geachtet werden.
- Typischerweise enthalten rekursive Prozeduren eine Abfrage, deren Ergebnis entscheidet, ob weitere rekursive Aufrufe stattfinden oder nicht.

Rekursiv vs iterativ

Factorial2.om

```
PROCEDURE Factorial(n: INTEGER) : INTEGER;
   VAR
      result: INTEGER;
BEGIN
   result := 1;
   WHILE n > 0 DO
      result := result * n; DEC(n);
   END;
   RETURN result
END Factorial;
```

- Viele Probleme lassen sich mindestens genauso elegant und deutlich effizienter auch iterativ lösen.
- Dies gilt insbesondere in Fällen von *tail recursion*, die sofort durch eine Schleife ersetzt werden können, da kein “verschachtelter Zustand” existiert, der durch eine Rekursion repräsentiert werden müßte.
- Bei einer *tail recursion* erfolgt der rekursive Aufruf (wenn überhaupt) nur ganz am Schluß:

```
PROCEDURE TailRecursion(...);
BEGIN
   IF (* einfacher Fall *) THEN
      (* ... *)
   ELSE
      (* ... *)
      TailRecursion(...);
   END;
END TailRecursion;
```

Rekursiv vs iterativ

Ackermann.om

```
PROCEDURE Ackermann(x, y: INTEGER) : INTEGER;
BEGIN
   IF x = 0 THEN
      RETURN y + 1
   ELSIF y = 0 THEN
      RETURN Ackermann(x - 1, 1)
   ELSE
      RETURN Ackermann(x - 1, Ackermann(x, y - 1))
   END;
END Ackermann;
```

- Es gibt berechenbare Funktionen, deren Rekursionstiefe sich nicht ohne die Berechnung derselben ermitteln läßt.

- Dazu gehört die obige Funktion von Wilhelm Ackermann in ihrer durch Rósa Péter und Raphael Robinson vereinfachten Form.

- Natürlich läßt sich die Ackermann-Funktion ohne den Selbstaufruf einer Prozedur lösen, aber dann muß die Rekursion "per Hand" nachprogrammiert werden.

- Mehr zum Unterschied zwischen *primitiv rekursiven* Funktionen (LOOP-berechenbar) und der allgemeineren Klasse der WHILE-berechenbaren Funktionen gibt es zum Beispiel im empfehlenswerten Buch von Uwe Schöning, "Theoretische Informatik kurz gefaßt".

Rekursiv vs iterativ

- Rekursion vs Iteration ist kein echter Gegensatz, da jede Art von Rekursion in einem Algorithmus iterativ abgearbeitet wird:

- Entweder beginnen wir mit einem Anfangszustand und leiten mit der Rekursions-Regel jeweils neue Zustände ab oder

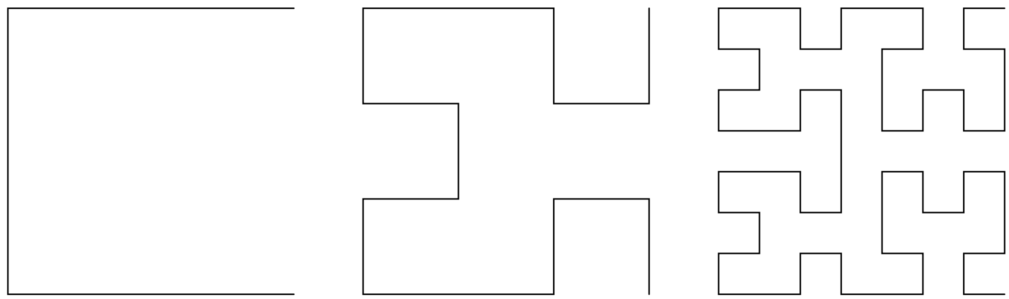
- wir beginnen mit einer komplexen Situation und vereinfachen sie sukzessive durch Anwendung der Rekursions-Regel, bis ein Trivialfall übrig bleibt, der sich ohne Anwendung von Rekursion erledigen läßt.

- Die Entscheidung, ob ein Problem mit sich selbst (direkt oder indirekt) aufzurufenden Prozeduren erledigt wird oder durch die Verwendung von Schleifen und ggf. zugehörigen Datenstrukturen, hängt im wesentlichen davon ab, wie

  - lesbar und verstehbar die alternativen Varianten sind und

  - welcher Rechen- und Speicheraufwand damit jeweils verbunden ist.

Peano-Hilbert-Kurven



- Die von Peano und Hilbert 1890/1891 entdeckten Kurven konvergieren gegen eine Funktion, die das Intervall [0,1] der reellen Zahlen in die Fläche [0,1] x [0,1] surjektiv (aber nicht bijektiv) abbildet und dabei stetig (jedoch nirgendwo differenzierbar) ist.

- 1968 fand Aristid Lindenmayer – ein Biologe, der auf der Suche nach mathematischen Modellen für die Entwicklung von Pflanzen war – ein System von rekursiven Ersetzungsregeln, das die Beschreibung dieser Kurven erlaubt:

  - Ausgangstext: `"L"`

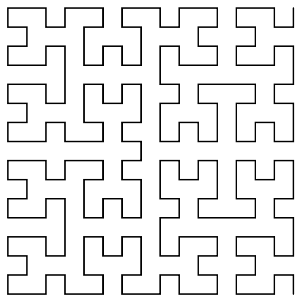
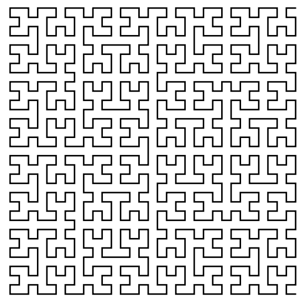
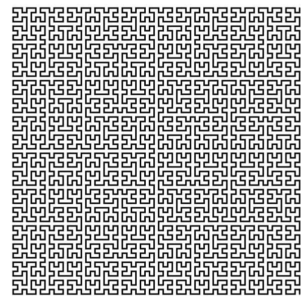
  - `"L" -> "+RF-LFL-FR+"`

  - `"R" -> "-LF+RFR+FL-"`

- Dabei wird von einer Turtle-Graphik ausgegangen:

`"+"`	Drehung nach links (hier um 90 Grad)
`"-"`	Drehung nach rechts (hier um 90 Grad)
`"F"`	Vorwärts bewegen und eine Linie zeichnen (in einer Einheitslänge)

Peano-Hilbert-Kurven

- Die Zeichenanweisungen für eine Kurve vom n-ten Grad ergeben sich dann aus der n-maligen Anwendung der Ersetzungsregeln:

Grad 0	`"L"`
Grad 1	`"+RF-LFL-FR+"`
Grad 2	`"+-LF+RFR+FL-F-+RF-LFL-FR+F+RF-LFL-FR+-F-LF+RFR+FL-+"`
Grad 3	`"+-+RF-LFL-FR+F+-LF+RFR+FL-F-LF+RFR+FL-+F+RF-LFL-FR+"`
	`"-F-+-LF+RFR+FL-F-+RF-LFL-FR+F+RF-LFL-FR+-F-LF+RFR+F"`
	`"L-+F+-LF+RFR+FL-F-+RF-LFL-FR+F+RF-LFL-FR+-F-LF+RFR+"`
	`"FL-+-F-+RF-LFL-FR+F+-LF+RFR+FL-F-LF+RFR+FL-+F+RF-LF"`
	`"L-FR+-+"`

- Bei den Zeichenanweisungen sind dann die rekursiven Aufrufe zu ignorieren, so daß dann jeweils folgendes übrig bleibt:

Grad 0	`""`
Grad 1	`"+F-F-F+"`
Grad 2	`"+-F+F+F-F-+F-F-F+F+F-F-F+-F-F+F+F-+"`
Grad 3	`"+-+F-F-F+F+-F+F+F-F-F+F+F-+F+F-F-F+-F-+-F+F+F-F-+F-"`
	`"F-F+F+F-F-F+-F-F+F+F-+F+-F+F+F-F-+F-F-F+F+F-F-F+-F-"`
	`"F+F+F-+-F-+F-F-F+F+-F+F+F-F-F+F+F-+F+F-F-F+-+"`

Peano-Hilbert-Kurven

Hilbert.om

```
PROCEDURE GenCurve(graphic: TurtleGraphics.Graphic;
                   level: INTEGER);

   (* Peano-Hilbert curves:
      start: L
      L -> +RF-LFL-FR+
      R -> -LF+RFR+FL-
   *)

   (* ... Forward, Left, and Right ... *)

   PROCEDURE ^ R(n: INTEGER);

   PROCEDURE L(n: INTEGER);
   BEGIN
      IF n > 0 THEN
         DEC(n);
         Left; R(n); Forward; Right; L(n); Forward;
         L(n); Right; Forward; R(n); Left;
      END;
   END L;

   PROCEDURE R(n: INTEGER);
   BEGIN
      IF n > 0 THEN
         DEC(n);
         Right; L(n); Forward; Left; R(n); Forward;
         R(n); Left; Forward; L(n); Right;
      END;
   END R;

BEGIN L(level);
END GenCurve;
```

Peano-Hilbert-Kurven

Hilbert.om

```
PROCEDURE L(n: INTEGER);
   (* L -> +RF-LFL-FR+ *)
BEGIN
   IF n > 0 THEN
      DEC(n);
      Left; R(n); Forward; Right; L(n); Forward;
      L(n); Right; Forward; R(n); Left;
   END;
END L;
```

- Die Ersetzungsregeln können unmittelbar übernommen werden.
- Jede der rekursiven Prozeduren erhält dabei einen Parameter n, der der noch fehlenden Rekursionstiefe entspricht.
- Bei $n = 0$ wird auf die Ausgabe von Anweisungen verzichtet.
- Da sich `L` und `R` wechselseitig aufrufen, muß eine der beiden Prozedur vordeklariert werden:

Hilbert.om

```
PROCEDURE ^ R(n: INTEGER);
```

Turtle-Graphiken

TurtleGraphics.od

```
DEFINITION TurtleGraphics;

   IMPORT Plotters, Services, Streams;

   TYPE
      Graphic = POINTER TO GraphicRec;
      GraphicRec = RECORD (Services.ObjectRec) END;

   PROCEDURE Create(VAR graphic: Graphic; angle: REAL);

   PROCEDURE EnableRoundCorners(graphic: Graphic;
                                cornerFraction: REAL);

   PROCEDURE PenDown(graphic: Graphic);
   PROCEDURE PenUp(graphic: Graphic);

   PROCEDURE Save(graphic: Graphic);
   PROCEDURE Restore(graphic: Graphic);

   PROCEDURE Left(graphic: Graphic);
   PROCEDURE Forward(graphic: Graphic);
   PROCEDURE Right(graphic: Graphic);

   PROCEDURE LeftBy(graphic: Graphic; angle: REAL);
   PROCEDURE RightBy(graphic: Graphic; angle: REAL);

   PROCEDURE ApplySymbols(graphic: Graphic;
                          symbols: Streams.Stream);

   PROCEDURE Plot(graphic: Graphic;
                  plotter: Plotters.Plotter);

END TurtleGraphics.
```

Turtle-Graphiken

- Turtle-Graphiken kamen auf mit der Programmiersprache Logo, die 1967 am MIT entstand (Seymour Papert, Wallace Feurzeig und andere).

- Bei Turtle-Graphiken gibt es keine absoluten Positionierungen in einem Koordinatensystem. Stattdessen sind alle Anweisungen relativ zu der aktuellen Position und Richtung zu sehen.

- Die Ausgabe erfolgt über die Schnittstelle von *Plotters*, für die es mehrere Implementierungen gibt:

  Plot5Streams: Erzeugt eine Ausgabe im sogenannten *plot(5)*-Format, das mit verschiedenen Werkzeugen in andere Formate konvertiert werden kann, z.B. mit */usr/lib/lp/postscript/postplot* in PostScript.

  XPlotters: Die Ausgabe erfolgt über X-Windows am Bildschirm.

Turtle-Graphiken

Hilbert.om

```
PROCEDURE GenCurve(graphic: TurtleGraphics.Graphic;
                   level: INTEGER);

   PROCEDURE Left;
   BEGIN TurtleGraphics.Left(graphic);
   END Left;

   PROCEDURE Right;
   BEGIN TurtleGraphics.Right(graphic);
   END Right;

   PROCEDURE Forward;
   BEGIN TurtleGraphics.Forward(graphic);
   END Forward;

   (* ... *)

END GenCurve;
```

- Um die rekursiven Prozeduren `L` und `R` möglichst kompakt zu halten, werden die Operationen `Left`, `Right` und `Forward` definiert, die die entsprechenden Prozeduren aus dem Modul *TurtleGraphics* aufrufen.

Turtle-Graphiken

Hilbert.om

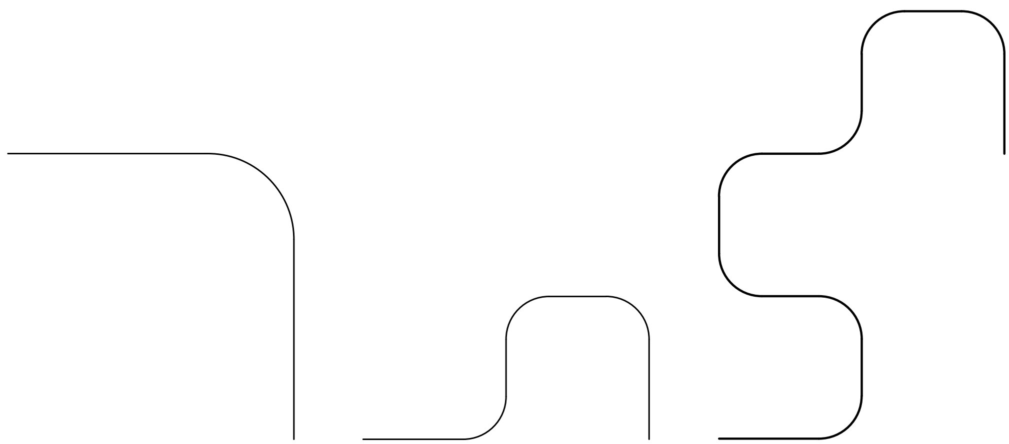
```
MODULE Hilbert;

   IMPORT Args := UnixArguments, Conclusions, Errors,
      Plot5Streams, Plotters, Read, RelatedEvents, Streams,
      TurtleGraphics, UnixFiles, XPlotters;

   VAR
      level: INTEGER;
      graphic: TurtleGraphics.Graphic;
      plotter: Plotters.Plotter;

   PROCEDURE ProcessArgs;
      (* ... *)
   END ProcessArgs;

   PROCEDURE GenCurve(graphic: TurtleGraphics.Graphic;
                      level: INTEGER);
      (* ... *)
   END GenCurve;

BEGIN
   ProcessArgs;
   TurtleGraphics.Create(graphic, 90);
   GenCurve(graphic, level);
   TurtleGraphics.Plot(graphic, plotter);
END Hilbert.
```

Turtle-Graphiken

Hilbert.om

```
PROCEDURE ProcessArgs;
   VAR
      s: Streams.Stream;
      flag: CHAR;
      outfile: ARRAY 512 OF CHAR;
      errors: RelatedEvents.Object;
      out: Streams.Stream;
BEGIN NEW(errors); RelatedEvents.QueueEvents(errors);
   out := NIL;
   Args.Init("[-o outfile] level");
   WHILE Args.GetFlag(flag) DO
      CASE flag OF
      | "o":   Args.FetchString(outfile);
               IF ~UnixFiles.Open(out, outfile,
                     UnixFiles.write + UnixFiles.create,
                     Streams.onebuf, errors) THEN
                  Conclusions.Conclude(errors,
                     Errors.fatal, "");
               END;
      ELSE
         Args.Usage;
      END;
   END;
   Args.Fetch(s); Read.IntS(s, level); Args.AllArgs;
   IF out # NIL THEN
      Plot5Streams.Create(plotter, out);
   ELSE
      IF ~XPlotters.Create(plotter, errors) THEN
         Conclusions.Conclude(errors, Errors.fatal, "");
      END;
   END;
END ProcessArgs;
```

Drachenkurven



- Drachen-Kurven wurden von drei Physikern bei der NASA entdeckt (John E. Heighway, Bruce A. Banks, and William G. Harter) und zuerst von Martin Gardner in Scientific American im März und April 1967 veröffentlicht.

- Die Ecken wurden durch Abrundungen ersetzt, damit die Kurve sich selbst nicht überschneidet.

- Ersetzungsregeln:

  - Ausgangstext: `"FX"`

  - `"X" -> "X+YF+"`

  - `"Y" -> "-FX-Y"`

- Zeichenanweisungen:

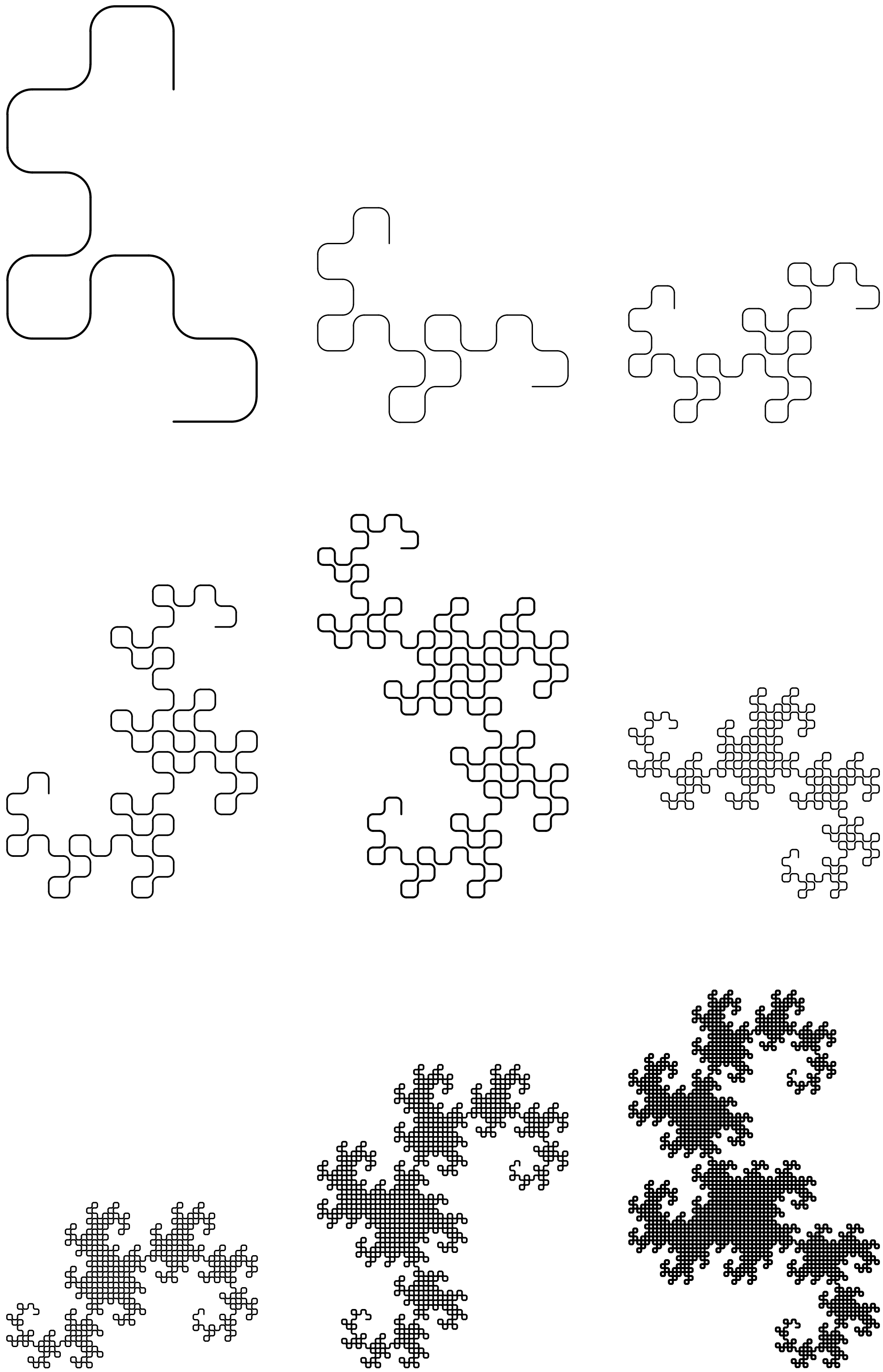
Grad 1	`"F+F+"`
Grad 2	`"F+F++-F-F+"`
Grad 3	`"F+F++-F-F++-F+F+--F-F+"`

Drachenkurven

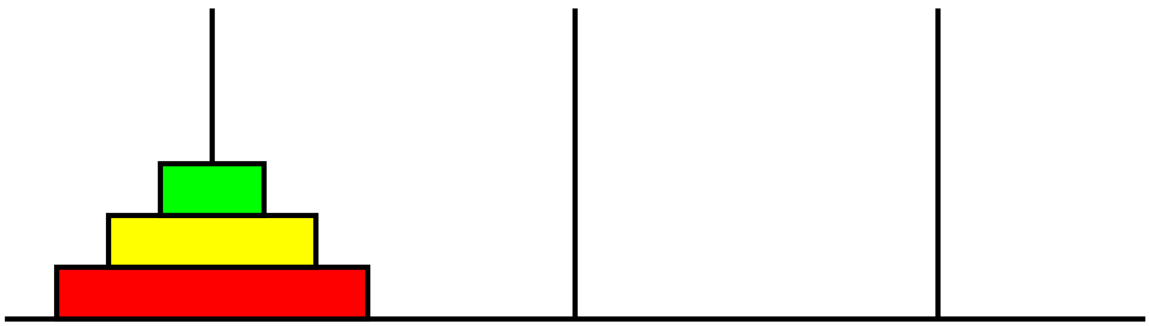
Dragon.om

```
PROCEDURE Dragon(graphic: TurtleGraphics.Graphic;
                 level: INTEGER);
   (* Dragon curve (X -> X+YF+, Y -> -FX-Y) *)

   PROCEDURE Left;
   BEGIN TurtleGraphics.Left(graphic);
   END Left;

   PROCEDURE Right;
   BEGIN TurtleGraphics.Right(graphic);
   END Right;

   PROCEDURE Forward;
   BEGIN TurtleGraphics.Forward(graphic);
   END Forward;

   PROCEDURE ^ Y(n: INTEGER);

   PROCEDURE X(n: INTEGER);
   BEGIN
      IF n > 0 THEN
         X(n-1); Left; Y(n-1); Forward; Left;
      END;
   END X;

   PROCEDURE Y(n: INTEGER);
   BEGIN
      IF n > 0 THEN
         Right; Forward; X(n-1); Right; Y(n-1);
      END;
   END Y;

BEGIN Forward; X(level);
END Dragon;
```

Drachenkurven



Türme von Hanoi



- Gegeben sind drei Plätze zum Stapeln und n Scheiben, die zu Beginn alle auf dem 1. Stapel liegen.

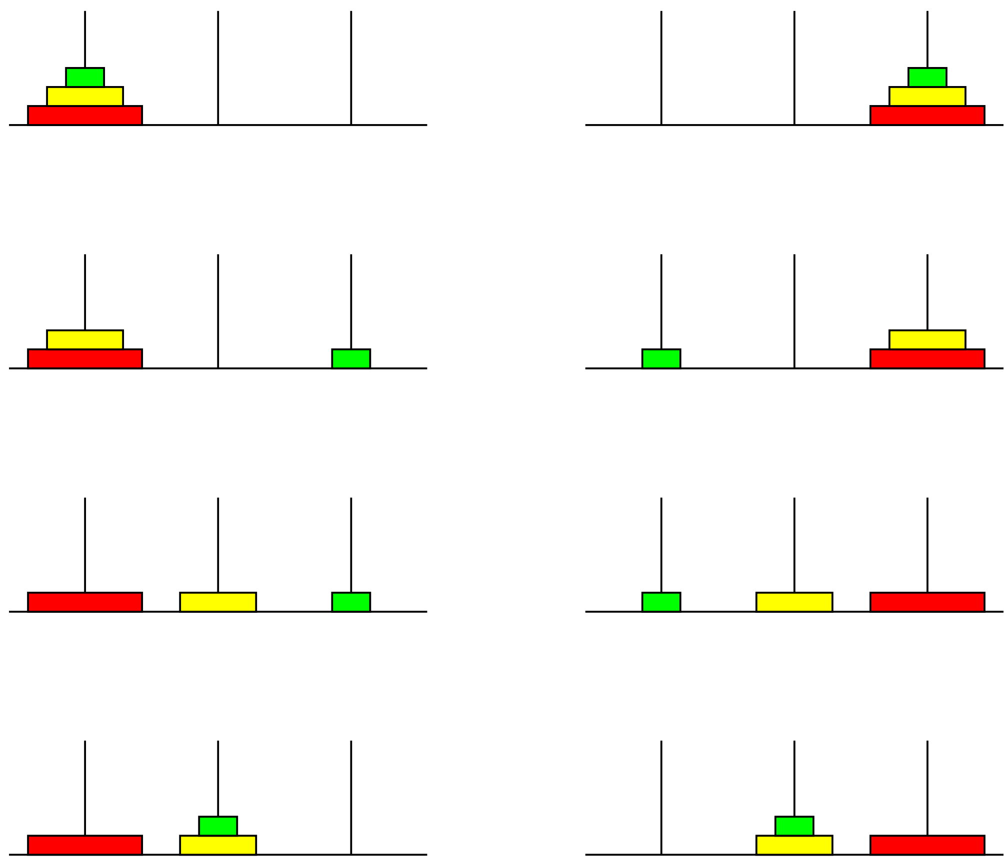
- Alle Scheiben sind unterschiedlich groß und sie müssen auf einem Stapel in geordneter Weise liegen, so daß nicht größere Scheiben auf kleineren zu liegen kommen.

- Jede Scheibe ist beweglich und kann von einem Stapel auf einen anderen getragen werden. Es dürfen jedoch nicht mehrere Scheiben auf einmal bewegt werden.

- Aufgabenstellung: Alle Scheiben sind von dem 1. Stapel auf den 3. zu befördern.

- Das Problem wurde 1883 von Edouard Lucas erfunden.

Türme von Hanoi



Türme von Hanoi

- Das Problem, n Scheiben vom Stapel a zum Stapel b zu befördern, läßt sich lösen, wenn

  - zunächst die obersten $n-1$ Scheiben von Stapel a zum als Hilfsstapel genutzten Stapel c verlegt werden,

  - dann die jetzt zuoberst liegende Scheibe auf a nach b verlegt wird und abschließend

  - wieder $n-1$ Scheiben vom Hilfsstapel c nach b verlagert werden.

- Zu beachten ist dabei, daß die Rollen der drei Stapel (Ausgangs-, Ziel- und Hilfsstapel) ständig wechseln.

- Insgesamt werden $2^n - 1$ Züge benötigt. Dies ist minimal.

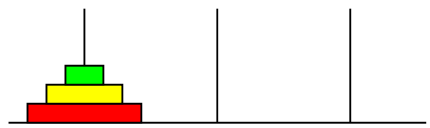
Hanoi.om

```
PROCEDURE MoveDisks(n: INTEGER; from, to, help: Tower);
BEGIN
   IF n > 0 THEN
      MoveDisks(n - 1, from, help, to);
      MoveDisk(from, to);
      MoveDisks(n - 1, help, to, from);
   END;
END MoveDisks;
```

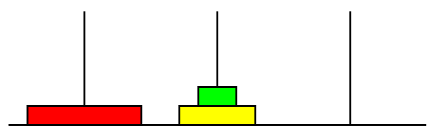
Türme von Hanoi

Situationen beim Aufruf von `MoveDisks(3, 1, 3, 2)`:

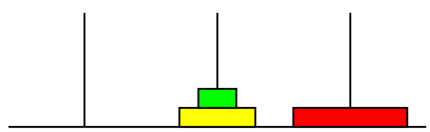
- Ausgangssituation:



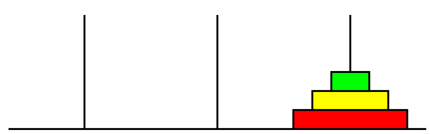
- Situation, nachdem 2 Scheiben vom Ausgangsstapel 1 zum Hilfsstapel 2 verlegt worden sind:



- Nach dem Verlegen der obersten Scheibe von 1 nach 3:



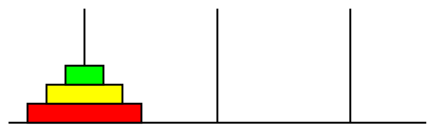
- Schlußsituation, nachdem 2 Scheiben von dem Stapel 2 nach 3 befördert sind:



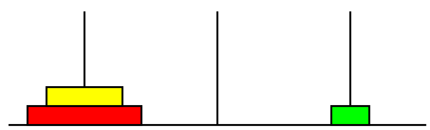
Türme von Hanoi

Situationen beim Aufruf von `MoveDisks(2, 1, 2, 3)`:

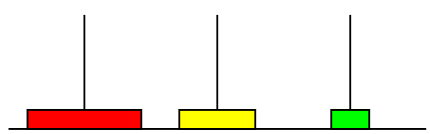
- Ausgangssituation:



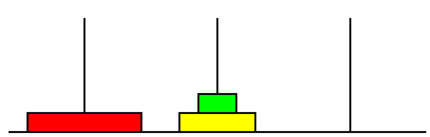
- Situation, nachdem eine Scheibe vom Ausgangsstapel 1 zum Hilfsstapel 3 verlegt worden ist:



- Nach dem Verlegen der obersten Scheibe von 1 nach 2:



- Schlußsituation, nachdem eine Scheibe von dem Stapel 3 nach 2 befördert ist:



Permutationen

- Eine Permutation von n Objekten ist eine Aneinanderreihung von n verschiedenen Objekten in eine Zeile.
- Es gibt 6 Permutationen für drei Objekte a, b, c:

$$abc, \qquad acb, \qquad bac, \qquad bca, \qquad cab, \qquad cba$$

- Es gibt $n!$ verschiedene Permutationen für n Objekte.

Permutationen

Problemstellung: Generierung aller Permutationen von n Objekten, die wir der Einfachheit wegen mit den Zahlen 1...n identifizieren. Knuth stellt im Abschnitt 1.2.5 zwei Methoden vor, die jeweils vorgeben, wie aus den vorgegebenen Permutationen für $n-1$ Objekte die möglichen Permutationen mit n Objekten generiert werden:

Methode 1: Für jede Permutation $a_1\ a_2\ ...\ a_{n-1}$ erzeuge n neue, indem die Zahl n in alle möglichen Plätze eingefügt wird:

$$n\ a_1\ a_2\ ...\ a_{n-1}, \qquad a_1\ n\ a_2\ ...\ a_{n-1} \qquad ..., \qquad a_1\ a_2\ ...\ a_{n-1}\ n$$

Methode 2: An jede Permutation $a_1 a_2 ... a_{n-1}$ wird eine Zahl k für $1 <= k <= n$ angehängt und jeweils jedes a_i um 1 erhöht, falls es $>= k$ ist.

Aus der Permutation 231 erhalten wir so durch das Anhängen von 1, 2, 3 und 4 die Permutationen

3 4 2 1, 3 4 1 2, 2 4 1 3, 2 3 1 4

Das zweite Verfahren ist vorteilhafter, wenn die Permutationen in einem Array abgelegt sind, da dann das Verschieben von Teilen des Arrays entfällt.

Permutationen

Permutations.om

```
PROCEDURE GenPermutations(k, n: INTEGER; perm: Permutation);
   (* perm[0..k-1] is already fixed,
      generate all variants for perm[k..n-1]
   *)
   VAR
      nperm: Permutation;
      i, j: INTEGER;
BEGIN
   IF k = n THEN
      GenPermutation(perm, n);
   ELSE
      i := 1;
      WHILE i <= k+1 DO
         nperm := perm;
         j := 0;
         WHILE j < k DO
            IF nperm[j] >= i THEN
               INC(nperm[j]);
            END;
            INC(j);
         END;
         nperm[k] := i;
         GenPermutations(k + 1, n, nperm);
         INC(i);
      END;
   END;
END GenPermutations;
```

- Nachteil dieser Lösung: Die Permutationen werden 2x kopiert: Einmal bei der Parameterübergabe und einmal bei der Bestimmung jeder neuen Permutation.

Permutationen

Permutations2.om

```
PROCEDURE GenPermutations(k, n: INTEGER; perm: Permutation);
   (* perm[n-k..n-1] is fixed except that
      those of perm[n-k+1..n-1] are to be incremented by 1
      which are greater or equal to perm[n-k];
      generate all variants for perm[0..n-k-1]
   *)
   VAR
      inserted: INTEGER;
      i: INTEGER;
BEGIN
   IF k > 1 THEN
      inserted := perm[n-k];
      i := n-k+1;
      WHILE i < n DO
         IF perm[i] >= inserted THEN
            INC(perm[i]);
         END;
         INC(i);
      END;
   END;
   IF k = n THEN
      GenPermutation(perm, n);
   ELSE
      i := 1;
      WHILE i <= k+1 DO
         perm[n-k-1] := i;
         GenPermutations(k + 1, n, perm);
         INC(i);
      END;
   END;
END GenPermutations;
```

Quicksort

Rekursion kann auch dazu genutzt werden, ein Problem in zwei (oder mehrere) einfacher zu lösende Teilprobleme zu zerlegen. Dies wird dann solange fortgesetzt, bis die Teilprobleme trivial zu lösen sind. Das von C. A. Hoare 1962 veröffentlichte Quicksort-Verfahren basiert auf dieser Vorgehensweise.

Problemstellung: Gegeben ist ein Feld K der Länge n mit Schlüsseln K_1 ... K_n, die so zu permutieren sind, daß $K_1 \leq ... \leq K_n$ gilt.

Grenzfall: Wenn die Länge n 0 oder 1 beträgt, entfällt eine weitere Sortierung.

Zerlegung in zwei Teilprobleme: Ein beliebiger Schlüssel P (das sogenannte Pivotelement) aus K_1 ... K_n ist auszuwählen (beispielsweise K_1). Dann wird K so partitioniert, daß es i, j aus $1..n$ gibt, so daß gilt

- $i = j + 1$,
- $K_t \leq P \; \forall \; t \in 1 ... j$ und
- $K_t \geq P \; \forall \; t \in i ... n$.

Quicksort

QuickSort.om

```
PROCEDURE QuickSort(VAR items: Items;
                    first, last: INTEGER);
   (* sort items[first..last] *)
   VAR
      pivot: Item;
      i, j: INTEGER;
BEGIN
   IF first < last THEN
      (* select pivot element *)
      pivot := items[first];
      (* divide ... *)
      i := first; j := last;
      REPEAT
         WHILE CompareItems(items[i], pivot) < 0 DO
            INC(i);
         END;
         WHILE CompareItems(items[j], pivot) > 0 DO
            DEC(j);
         END;
         IF i <= j THEN
            SwapItems(items[i], items[j]);
            INC(i); DEC(j);
         END;
      UNTIL i > j;
      (* ... and conquer *)
      QuickSort(items, first, j);
      QuickSort(items, i, last);
   END;
END QuickSort;
```

- Diese Variante lehnt sich an die Version von Niklaus Wirth, “Algorithmen und Datenstrukturen”, Teubner-Verlag

Literaturhinweise zur Rekursion

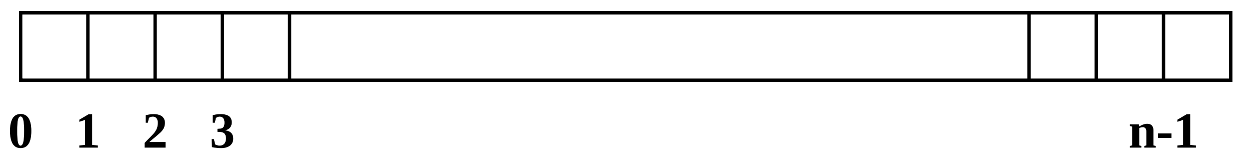
Da Rekursionen in so vielen Feldern vorkommen, ist es nicht sehr sinnvoll, hierfür eine auch nur annähernd abdeckende Aufzählung zu geben. Folgendes sind nur einige wenige Tips zur Vertiefung, deren Anregungen teilweise in die vorangegangenen Folien eingeflossen sind:

- Douglas R. Hofstadter, “Gödel, Escher, Bach – ein Endlos Geflochtenes Band”, Kapitel V, Rekursive Strukturen und Prozesse
- Uwe Schöning, “Theoretische Informatik kurz gefaßt”
- Günter Dotzel, “A Function to end all functions”, `http://www.modulaware.com/mdlt08.htm` liefert Hinweise zur effizienten Berechnung der Ackermann-Funktion
- Robert M. Dickau, “2D L-Systems”, `http://forum.swarthmore.edu/advanced/robertd/lsys2d.html`
- Alexander Bogomolny, “Plane Filling Curves”, `http://www.cut-the-knot.com/do_you_know/hilbert.html`
- Alexander Bogomolny, “Fractal Curves and Dimension”, `http://www.cut-the-knot.com/do_you_know/dimension.html`
- Alexander Bogomolny, “Tower of Hanoi”, `http://www.cut-the-knot.com/recurrence/hanoi.html`
- Donald E. Knuth, “The Art of Computer Programming”, Band 1, Abschnitt 1.2.5 über Permutationen

Zeiger

- Zeiger sind ein unverzichtbares Instrument für dynamische Datenstrukturen.
- Leider ist die Verwendung von Zeigern nicht trivial:
  - Techniken, die mit Zeigern arbeiten, sind schwer zu lesen. Es hilft hier nur Routine, die auf der Kenntnis aller gängigen Techniken beruht.
  - Die Fehlermöglichkeiten werden dramatisch erhöht durch das Potential undefinierter Zeiger. Die Mehrheit von Programmabstürzen geht darauf zurück.

Adreßraum



- Wenn ein Programm zur Ausführung gelangt, liegen Programmtext und Daten im Speicher.
- Der Speicher kann als ARRAY betrachtet werden, dessen einzelne Elemente Speicherzellen sind.
- Der Indexbereich dieses ARRAYs nennt sich Adreßraum, bei den einzelnen Speicherzellen handelt es sich heute üblicherweise um Bytes (zu 8 Bit).
- Unter UNIX (und anderen moderneren Betriebssystemen) hat jedes aktive Programm seinen eigenen virtuellen Adreßraum.
- In der Ulmer Oberon-Bibliothek wird der eigene Adreßraum von dem Modul *Memory* verwaltet.

Adreßraum unter UNIX

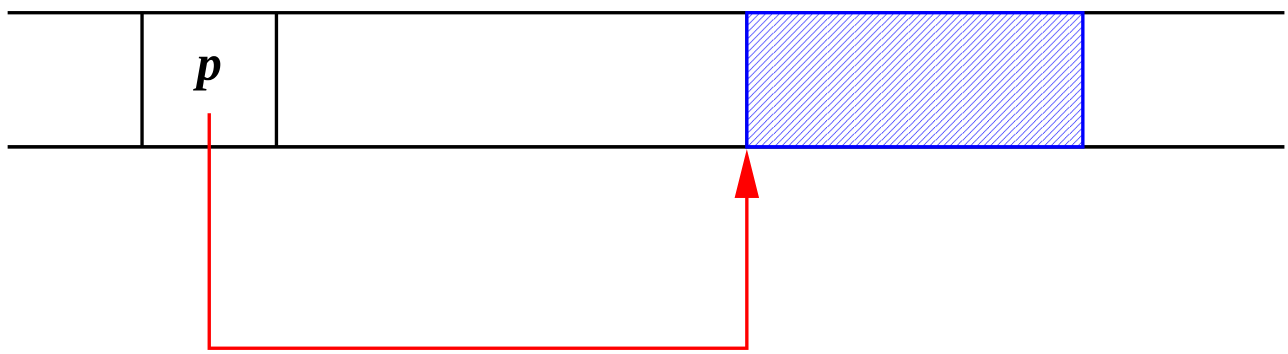
- Unter UNIX liegt im virtuellen Adreßraum eines Prozesses zu Beginn der Programmtext (Maschinencode), gefolgt von den globalen Variablen.

- Für lokale Variablen und Parameter gibt es einen Stapel (*stack*), der ganz am Ende des Adreßraumes beginnt und der (entsprechend der Verschachtelung der Prozeduraufrufe) von hohen zu immer niedriger werdenden Adressen wächst.

- Dem gegenüber, direkt hinter den globalen Variablen, werden unter UNIX üblicherweise dynamische Daten allokiert (*heap*).

```
thales$ /usr/proc/bin/pmap 71
71:     ttt
00010000    584K read/exec         dev:32,22 ino:1422140
000B0000     16K read/write/exec   dev:32,22 ino:1422140
000B4000    120K read/write/exec     [ heap ]
000D4000     32K read/write          [ anon ]
000F0000     64K read/write          [ anon ]
00102000     64K read/write          [ anon ]
00120000    128K read/write          [ anon ]
7FFFA000     16K read/write          [ anon ]
BFFFA000     16K read/write          [ anon ]
EFFFC000     16K read/write          [ stack ]
 total     1056K
thales$
```

Dynamische Daten



- Wenn weitere Speicherflächen zur Laufzeit benötigt werden, müssen sie an bisher freien Bereichen des Adreßraumes allokiert werden.

- Im Unterschied zu globalen Variablen sind jedoch dann die Adressen der Speicherlokationen nicht mehr konstant.

- Entsprechend werden Zeiger benötigt. Zeiger sind Variablen, die als Wert eine Adresse haben, die auf eine Speicherlokation verweist, an der die gewünschten Daten stehen.

Zeiger in Oberon

```
TYPE Record = RECORD (* ... *) END;
TYPE Pointer = POINTER TO Record;
VAR p: Pointer;
```

- Mit **POINTER TO** kann in Oberon ein Zeiger-Typ deklariert werden.
- Bei Zeiger-Typen muß immer der Typ angegeben werden, worauf verwiesen wird. Hierbei sind nur Records oder ARRAYs zulässig.
- Zeiger haben in Oberon immer einen wohldefinierten Wert. Dieser ist entweder **NIL** (d.h. sie zeigen auf kein Objekt) oder eine wohldefinierte Adresse auf ein lebendes Objekt des zugehörigen Typs.

Allokation in Oberon

```
TYPE
   BookRec =
      RECORD
         author, title: ARRAY 80 OF CHAR;
         year: INTEGER;
      END;
   Book = POINTER TO BookRec;

PROCEDURE CreateBook(VAR book: Book;
                     author, title: ARRAY OF CHAR;
                     year: INTEGER);
BEGIN
   NEW(book);
   COPY(author, book.author);
   COPY(title, book.title);
   book.year := year;
END CreateBook;
```

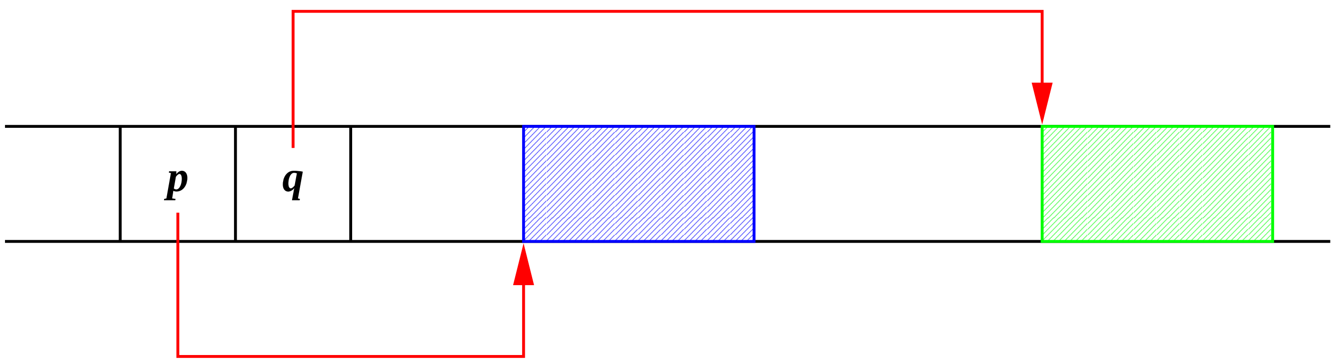
- Mit **NEW** wird eine Datenfläche allokiert und dem angegebenen Zeiger wird die Adresse auf die neu angelegte Datenfläche zugewiesen.

- Auf den Datentyp hinter dem Zeiger kann genauso zugegriffen werden als wäre die Zeigervariable von dem Typ, auf den sie verweist.

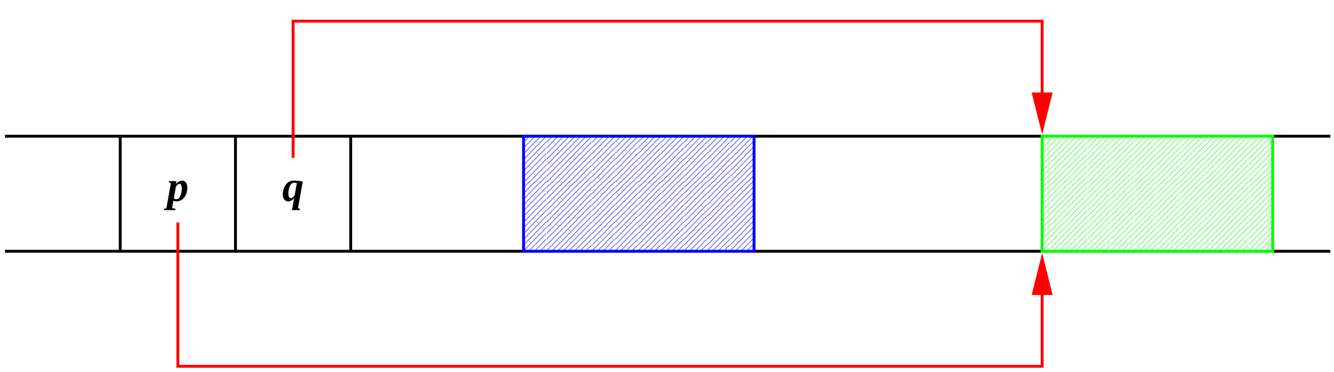
- Prozeduren, die einen Record allokieren, initialisieren und einen Zeiger darauf zurückliefern, nennen sich Konstruktoren.

- Wird versucht, auf Daten hinter einem Zeiger zuzugreifen, der den Wert **NIL** hat, so führt dies zu einem Laufzeitfehler.

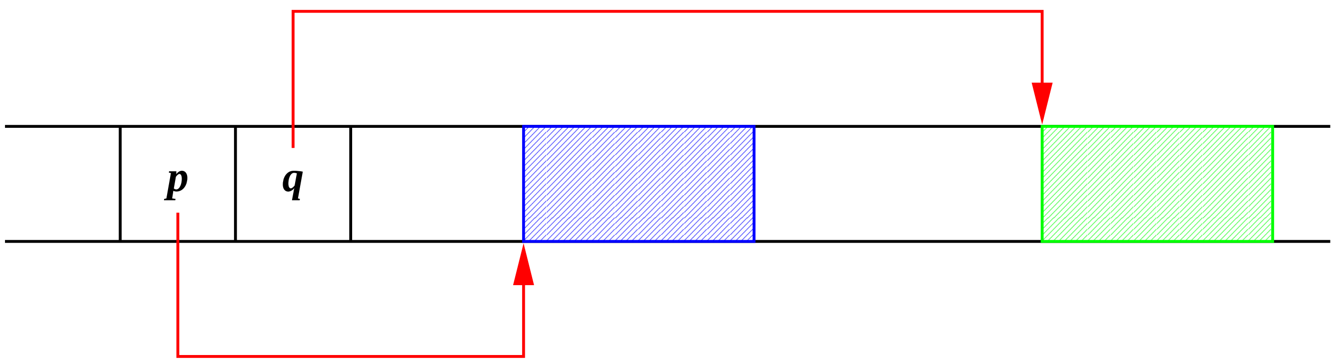
Zeigerzuweisung



- `p := q;`
- Bei einer Zeigerzuweisung werden keine Records (oder ARRAYs) kopiert, sondern nur der Wert einer Adresse.
- Wenn zwei (oder mehr) Zeiger auf die gleiche Datenfläche zeigen, wirkt dies, als würden mehrere Variablennamen für die gleichen Daten zur Verfügung stehen:
  `p.info := 1; q.info := 2; (* p.info = 2 *)`



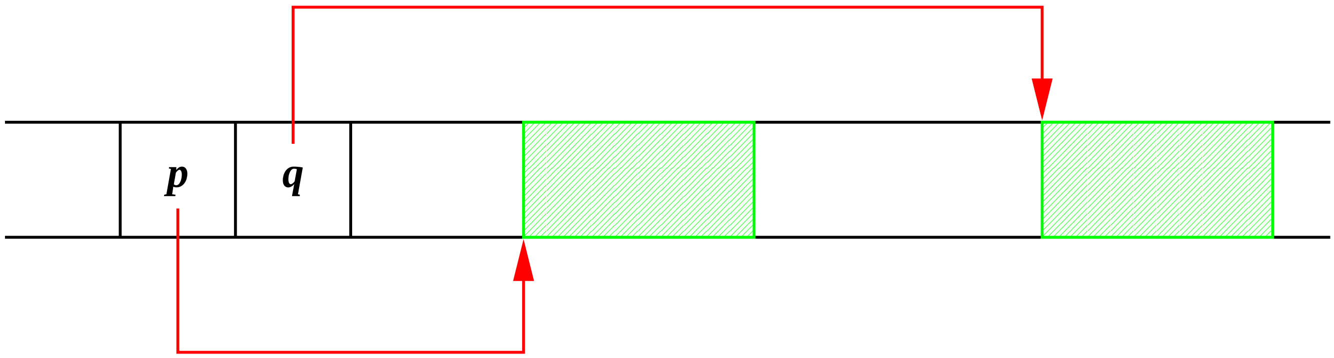
Record-Zuweisung über Zeiger



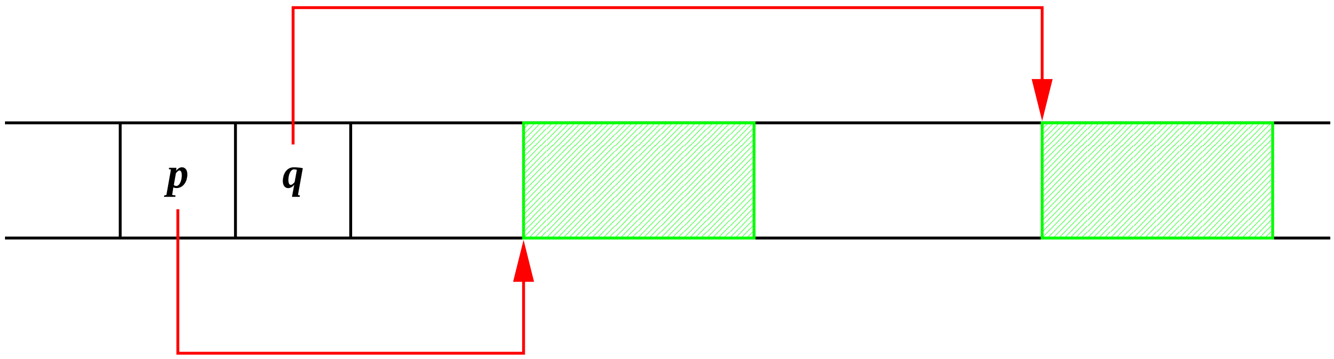
- `p^ := q^;`

- `p^` steht für den gesamten Record (bzw. das gesamte **ARRAY**), auf das *p* verweist.

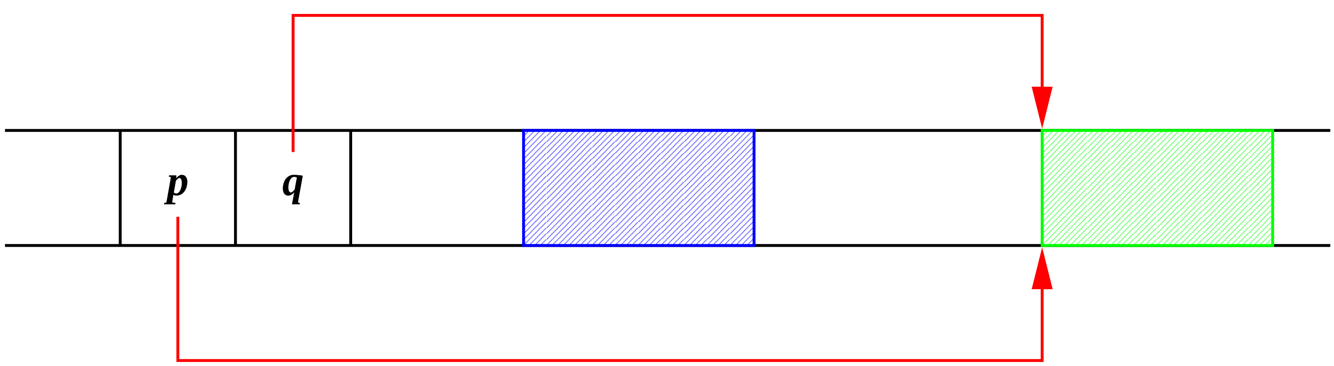
- Entsprechend lassen sich auch die vollständigen Datenflächen hinter einem Zeiger zuweisen. Dies geschieht jedoch eher selten, da typischerweise primär mit Zeigern gearbeitet wird (ist effizienter).



Vergleich von Zeigern



- `p = q`
- `p # q`
- Die Vergleichs-Operatoren `=` und `#` sind für Zeiger zulässig und beziehen sich auf die Adreßwerte und nicht auf die hinter den Zeigern liegenden Daten.
- So ist im obigen Beispiel `p # q` **TRUE**, während dies im unteren Falle für `p = q` zutrifft.



Übergabe von Zeigern als Parameter

```
PROCEDURE UpdateBook(book: Book);
   (* set book.year to current year *)
   VAR
      now: Times.Time;
      date: Dates.InfoRec;
BEGIN
   Clocks.GetTime(Clocks.system, now);
   Dates.Get(now, date);
   book.year := date.year;
END UpdateBook;
```

- Bei der Parameter-Übergabe ohne Zeiger ließ sich der Art der Parameterübergabe (**VAR**-Parameter vs Werte-Parameter) leicht entnehmen, ob der Parameter modifiziert werden kann (und wohl soll) oder nicht.

- Wenn Zeiger als Werte-Parameter übergeben werden, können trotzdem jederzeit die Daten hinter dem Zeiger modifiziert werden (nur der Zeiger kann nicht auf ein anderes Objekt gesetzt werden).

- Zeiger sind fast nur bei Konstruktoren und Suchoperationen als **VAR**-Parameter zu beobachten.

Konventionen bei Zeigern

```
TYPE
   BookRec =
      RECORD
         author, title: ARRAY 80 OF CHAR;
         year: INTEGER;
      END;
   Book = POINTER TO BookRec;
```

- Entscheiden Sie bei jedem Record-Typ ein für alle Mal, ob Sie ihn direkt benutzen oder nur indirekt über Zeiger. Gemischte Anwendungen erschweren die Lesbarkeit erheblich.

- Wenn Sie sich dafür entscheiden, auf einen bestimmten Record-Typ nur über Zeiger zuzugreifen, dann sollte der Zeiger-Typ unmittelbar bei der Record-Definition stehen.

- Dem Zeiger-Typ wird dann der kurze bündige Name (*ohne* Endungen wie "Ptr" oder "Pointer") gegeben, während der (dann nur noch extrem selten benötigte) Name des Record-Typs eine Endung erhält (z.B. "Rec" oder "Desc").

- Wenn eine Zeiger-Variable ihre Bindung zu einem Objekt nicht wechselt (z.B. der Parameter *book* bei der Prozedur *UpdateBook*), dann sollte ein Variablen-Namen verwendet werden, der dem Datentyp entspricht, auf den er zeigt (z.B. *book* und **nicht** *bookptr*).

- Nur Zeiger-Variablen, die ständig auf andere Objekte zeigen, dürfen ihren Zeiger-Charakter im Variablennamen offenbaren.

Rekursive Datentypen

```
TYPE
   Person = POINTER TO PersonRec;
   PersonRec =
      RECORD
         name: ARRAY 80 OF CHAR;
         mother, father: Person;
      END;
```

- Das beliebige Anwachsen dynamischer Datenstrukturen geht nur über rekursive Datentypen.

- Ein Record kann auf sich selbst verweisen. Damit dies bei der Typ-Definition klappt, ist es zulässig, einen Zeiger zu einem Record-Typ zu definieren, der noch nicht bekannt ist.

Freigabe von Datenflächen

- Im Trivialfall wird nur dynamisch Speicher angelegt, jedoch nie wieder freigegeben (z.B. bei klassischen Pascal-Implementierungen). Dies ist einfach zu realisieren, begrenzt jedoch natürlich die Möglichkeiten, da nicht beliebig viel Speicher zur Verfügung steht, um dies lange genug durchzuhalten.

- Bei vielen klassischen Programmiersprachen wie Modula-2 und C erfolgt die Freigabe explizit, z.B. in Modula-2 über **DISPOSE**. Nachteile:

  - Die Freigabe kann vergessen werden (Speicherlecks),

  - es ist nicht immer trivial zu ermitteln, wann etwas freigegeben werden sollte, und

  - Zeiger werden noch weiterverwendet, obwohl die Datenfläche dahinter bereits freigegeben worden ist (*dangling reference*).

  Insbesondere letzteres führt zu extrem schwer aufzudeckenden Problemen.

- In Oberon (und anderen modernen Programmiersprachen) erfolgt die Freigabe vollautomatisch. Alle Datenflächen, auf die nicht mehr zugegriffen werden kann, werden bei Bedarf wiederverwendet (*garbage collection*).

- Beim Ulmer Oberon-System wird ein *copying garbage collector* verwendet, der Fragmentierung ausschließt.

Listen

- Datenstrukturen sind mehr als eine Anhäufung von Daten.

- Wenn Zugriffszeiten eine Rolle spielen, dann sollten die Daten strukturell so angeordnet werden, daß alle wichtigen Zugriffs-Operationen effizient unterstützt werden. Entsprechend spiegeln sich häufig die strukturellen Beziehungen der Daten und die Zugriffsmethoden auch in der Vernetzung der Daten wider.

- Entsprechend hängt die Wahl einer geeigneten Datenstruktur von der Art und Häufigkeit der benötigten Zugriffsoperationen ab.

- Zu den einfachsten Datenstrukturen zählen lineare Listen, bei denen eine Reihe von Elementen a_i in einer geordneten Form repräsentiert werden:

$$a_1 \; a_2 \; \ldots \; a_{i-1} \; a_i \; a_{i+1} \; \ldots \; a_n$$

- Zu den typischen Operationen auf linearen Listen zählen

  - das Hinzufügen eines Elements am Anfang oder oder am Ende der Liste,

  - das Entfernen eines Elements,

  - das Betrachten des i-ten Elements (insbesondere für $i = 1$ oder $i = n$) und

  - das Betrachten des nächsten oder vorherigen Elements (also a_{i+1} oder a_{i-1} ausgehend von a_i).

Beispiele für Listen

- Aufträge: Jeder Eintrag entspricht einem Auftrag, der angibt, welcher Artikel an wen zu versenden ist. Neue Aufträge werden an das Ende der Liste angefügt. Der erste Auftrag wird bearbeitet und nach Erledigung entfernt.

- Spielzüge: Jeder Eintrag entspricht einem Spielzug. Wenn normal weitergezogen wird, kommt ein neuer Spielzug hinzu, der an das Ende der Liste anzufügen ist. Wenn jedoch ein Spielzug zurückgenommen werden soll, wird der letzte Eintrag aus der Liste entfernt und dazu verwendet, den alten Spielstand zu restaurieren.

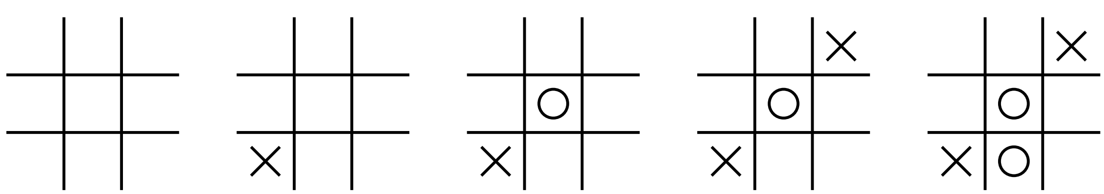
- Terminkalender: Jeder Eintrag besteht aus einer Zeitangabe und einer kurzen Information. Alle Einträge werden nach der Zeit sortiert angeordnet. Betrachtet wird nur der erste Eintrag, der als nächster relevant ist. Hinzukommende Einträge werden entsprechend ihrer Zeit einsortiert. Einträge am Anfang der Liste, die nicht mehr aktuell sind, werden entfernt.

Stapel

- Ein Stapel (oder Stack) ist eine lineare Liste, die sich dadurch auszeichnet, daß nur an einem der beiden Enden der Liste Elemente hinzugefügt und wieder weggenommen werden.

- Bei einem Stapel ist auch fast immer nur das jeweils oberste Element von Interesse, d.h. die “tieferliegenden” Elemente “tauchen” erst dann wieder auf, wenn alle darüberliegenden Elemente entfernt worden sind.

- So könnte ein Stapel mit Zuständen für Tic-Tac-Toe aussehen. Links ist das erste (ganz unten liegende) Element mit der Start-Situation. Im Verlaufe des Spieles kamen weitere Situationen hinzu. Wenn nun ein unzufriedener Spieler den letzten Zug rückgängig machen möchte, wird das Element jeweils ganz rechts entfernt:



Operationen für Stapel

```
PROCEDURE InitList(VAR list: List);
PROCEDURE AddElement(VAR list: List; element: Element);
PROCEDURE RemoveElement(VAR list: List);
PROCEDURE GetCurrentElement(list: List;
                            VAR element: Element);
PROCEDURE Length(list: List) : INTEGER;
PROCEDURE Full(list: List) : BOOLEAN;
```

- *AddElement* fügt das angegebene Element auf den Stapel oben hinzu.

- *RemoveElement* entfernt das oberste Element des Stapels.

- *GetCurrentElement* liefert das derzeitig oberste Element zurück.

- *Length* liefert die Anzahl der Elemente im Stapel zurück und *Full* gibt an, ob die maximale Kapazität des Stapels erreicht ist. So darf *AddElement* nur aufgerufen werden, wenn *Full* **FALSE** liefert, und *RemoveElement* und *GetCurrentElement* sind nur dann zulässig, falls *Length* > 0.

- Vielfach sind auch andere Namen für die Operationen gebräuchlich, populär sind insbesondere *Push* für *AddElement* und *Pop* für *RemoveElement*.

Stapel auf Basis von ARRAYs

TicTacToe.om

```
TYPE
   ListOfStates =
      RECORD
         state: ARRAY maxnofstates OF State;
         nofstates: SHORTINT; (* [0..maxnofstates] *)
      END;

PROCEDURE InitListOfStates(VAR list: ListOfStates);
BEGIN
   list.nofstates := 0;
END InitListOfStates;

PROCEDURE AddState(VAR list: ListOfStates; state: State);
BEGIN
   ASSERT(list.nofstates < maxnofstates);
   list.state[list.nofstates] := state;
   INC(list.nofstates);
END AddState;

PROCEDURE RemoveState(VAR list: ListOfStates);
BEGIN
   ASSERT(list.nofstates > 0);
   DEC(list.nofstates);
END RemoveState;
```

- Stapel können mit Hilfe von ARRAYs realisiert werden.
- Zusätzlich zum ARRAY wird noch ein Index benötigt, der auf das nächste freie Element zeigt bzw. den Füllgrad angibt.

Stapel auf Basis von ARRAYs

TicTacToe.om

```
PROCEDURE GetCurrentState(VAR list: ListOfStates;
                          VAR state: State);
BEGIN
   ASSERT(list.nofstates > 0);
   state := list.state[list.nofstates - 1];
END GetCurrentState;

PROCEDURE Length(list: ListOfStates) : SHORTINT;
BEGIN
   RETURN list.nofstates
END Length;
```

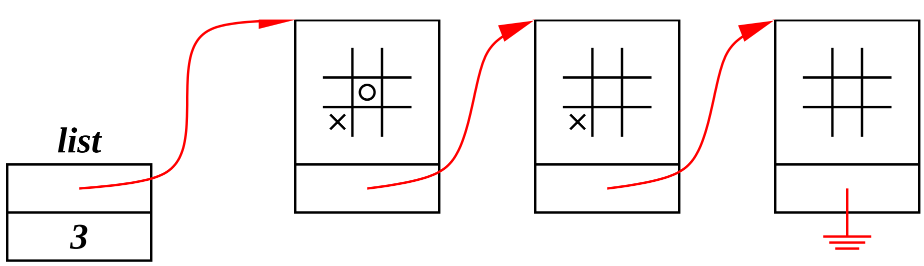
- Vorteil: Einfache Realisierung und auch ein Zugriff auf das i-te Element wäre effizient möglich.
- Nachteil: Die maximale Zahl der aufnehmbaren Elemente ist begrenzt und muß im voraus festgelegt werden.

Stapel auf Basis von verzeigerten Elementen

TicTacToe2.om

```
TYPE
   StateElement = POINTER TO StateElementRec;
   StateElementRec =
      RECORD
         state: State;
         next: StateElement;
      END;
   ListOfStates =
      RECORD
         top: StateElement;
         nofstates: INTEGER;
      END;
```

- Stapel lassen sich mit Hilfe von Zeigern realisieren.
- Jedes Element der Liste hat dann ein Feld, das auf das “darunterliegende” Element verweist.



Stapel auf Basis von verzeigerten Elementen

TicTacToe2.om

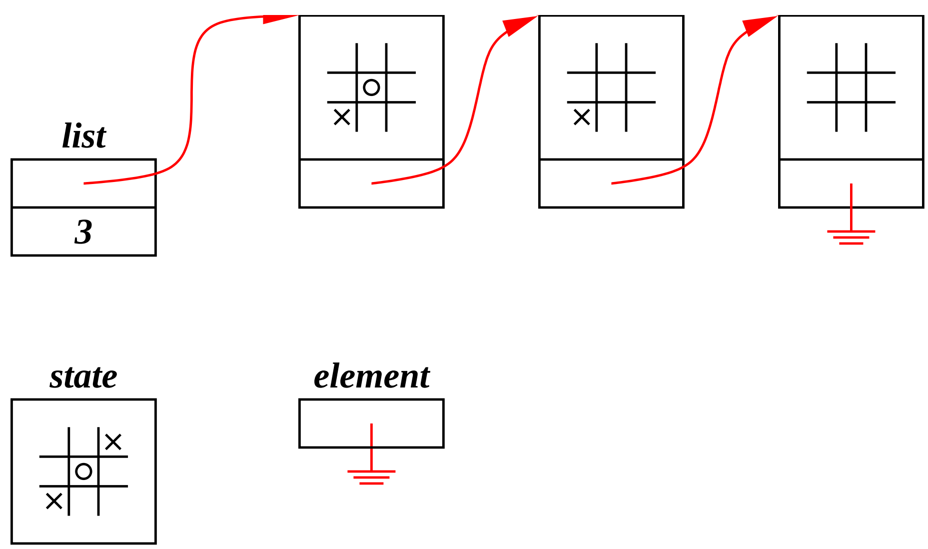
```
PROCEDURE InitListOfStates(VAR list: ListOfStates);
BEGIN
   list.top := NIL; list.nofstates := 0;
END InitListOfStates;

PROCEDURE AddState(VAR list: ListOfStates; state: State);
   VAR
      element: StateElement;
BEGIN
   NEW(element); element.state := state;
   element.next := list.top; list.top := element;
   INC(list.nofstates);
END AddState;

PROCEDURE RemoveState(VAR list: ListOfStates);
BEGIN
   ASSERT(list.top # NIL);
   list.top := list.top.next;
   DEC(list.nofstates);
END RemoveState;

PROCEDURE GetCurrentState(VAR list: ListOfStates;
                          VAR state: State);
BEGIN
   ASSERT(list.nofstates > 0);
   state := list.top.state;
END GetCurrentState;

PROCEDURE Length(list: ListOfStates) : INTEGER;
BEGIN
   RETURN list.nofstates
END Length;
```

Einfügen in eine verzeigerte Liste

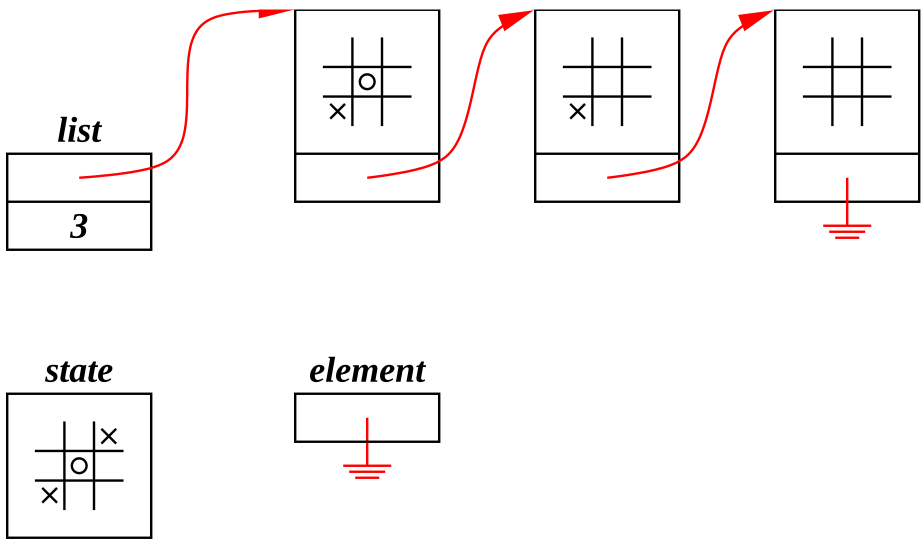
TicTacToe2.om

```
PROCEDURE AddState(VAR list: ListOfStates; state: State);
   VAR
      element: StateElement;
```

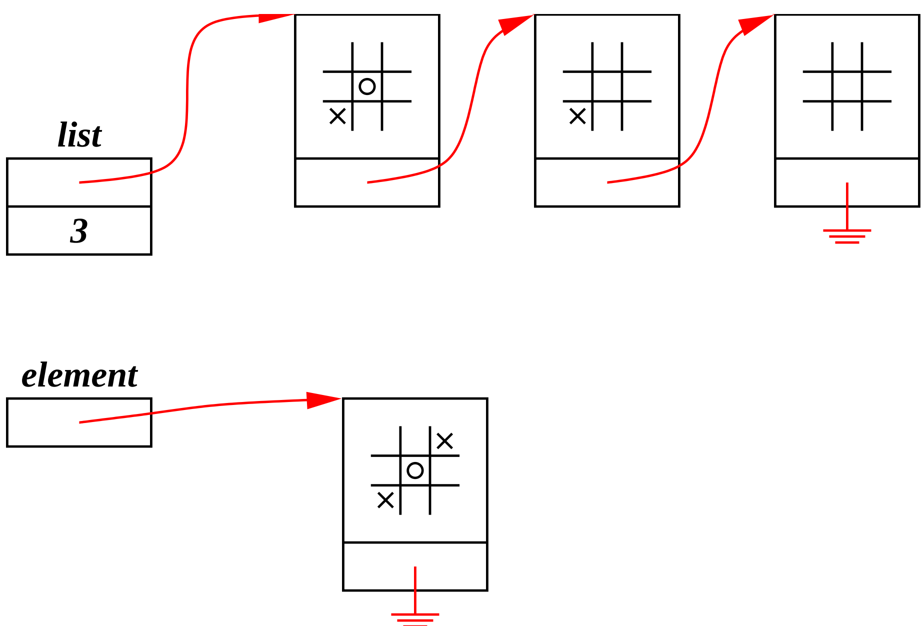
- Parameter und lokale Variablen zu Beginn:



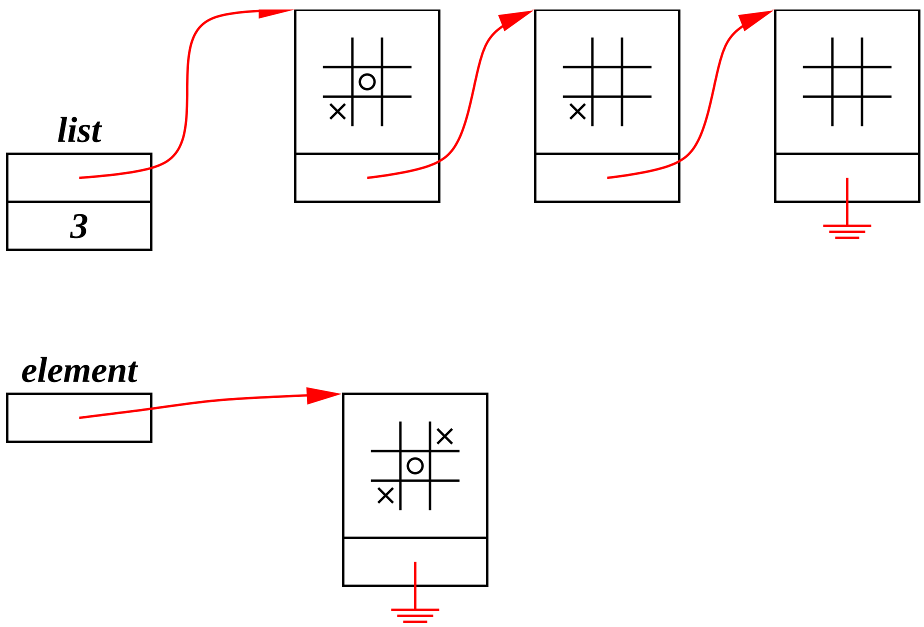
Einfügen in eine verzeigerte Liste



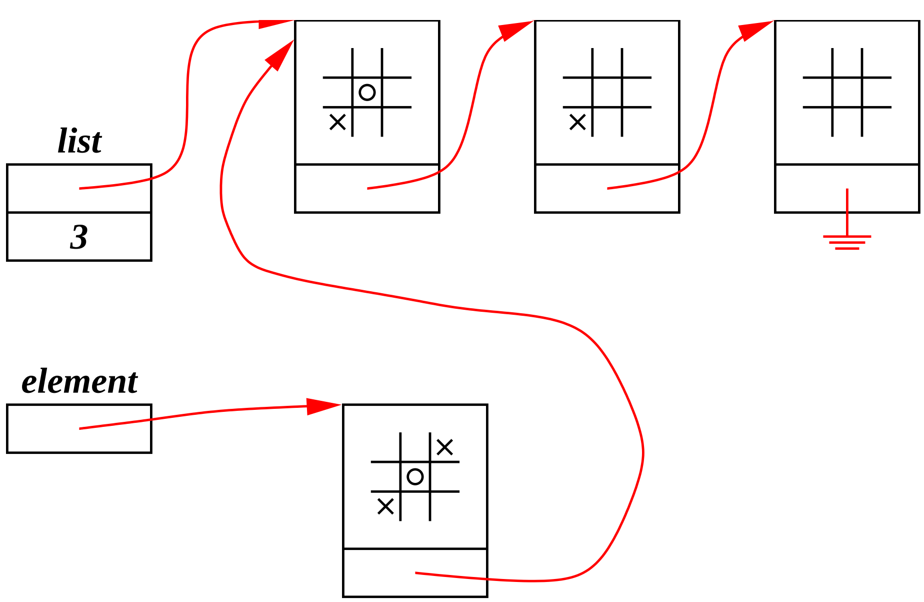
- `NEW(element); element.state := state;`



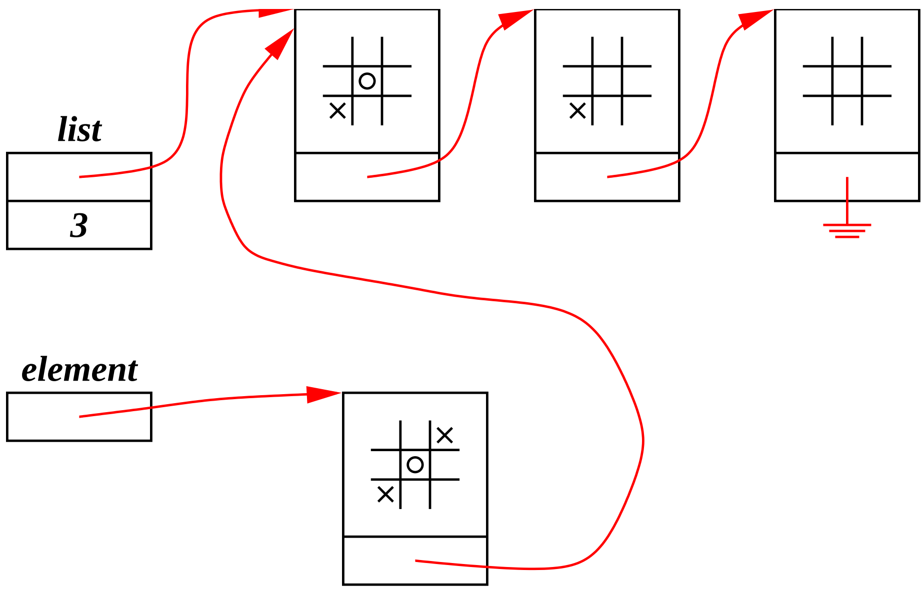
Einfügen in eine verzeigerte Liste



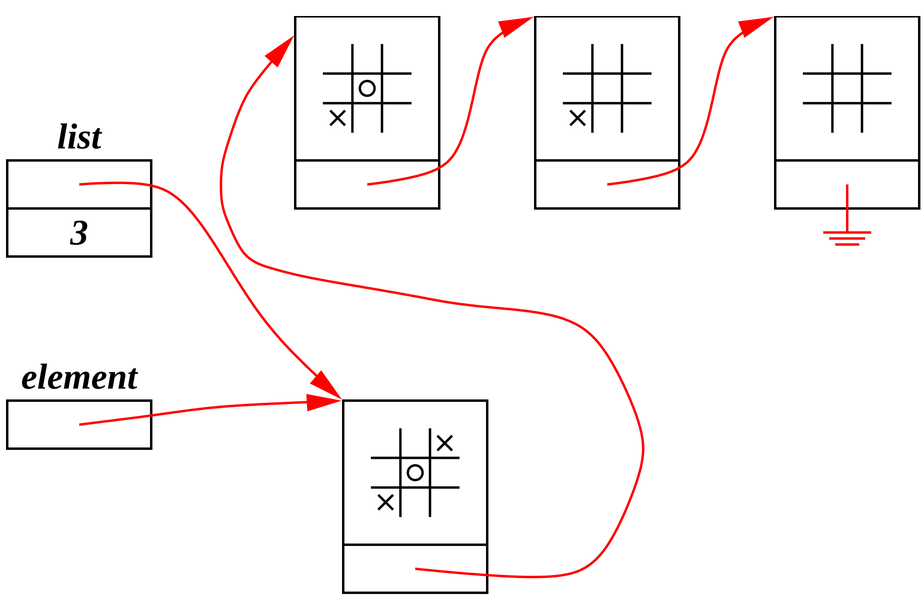
- element.next := list.top;



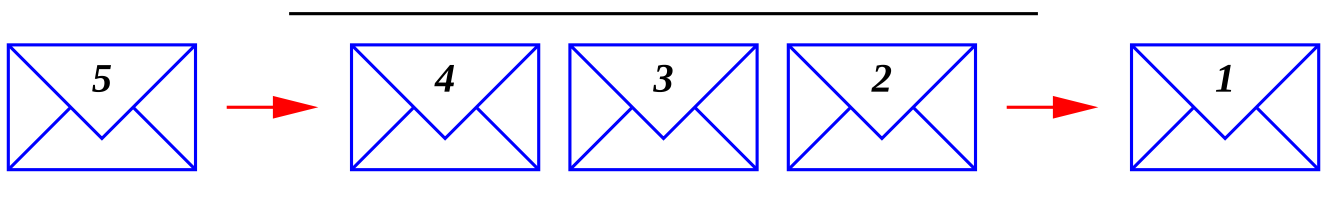
Einfügen in eine verzeigerte Liste



- list.top := element;



Warteschlangen



- Eine Warteschlange (alternative Namen: Queue und FIFO) ist eine lineare Liste, bei der an einem Ende Elemente hinzugefügt werden und an dem anderen Ende wieder weggenommen werden.

- Normalerweise werden bei einer Warteschlange die “innenliegenden” Objekte nicht betrachtet. Wesentlich ist nur, daß die Elemente in genau der Reihenfolge abgearbeitet werden, in der sie ursprünglich eingefügt worden sind.

Operationen für Warteschlangen

```
PROCEDURE InitQueue(VAR queue: Queue);
PROCEDURE AddElement(VAR queue: Queue; element: Element);
PROCEDURE RemoveElement(VAR queue: Queue;
                        VAR element: Element);
PROCEDURE Length(queue: Queue) : INTEGER;
PROCEDURE Full(queue: Queue) : BOOLEAN;
```

- *AddElement* hängt das angegebene Element an das Ende der Warteschlange.

- *RemoveElement* entfernt das Element der Liste mit der höchsten Verweildauer und liefert es zurück.

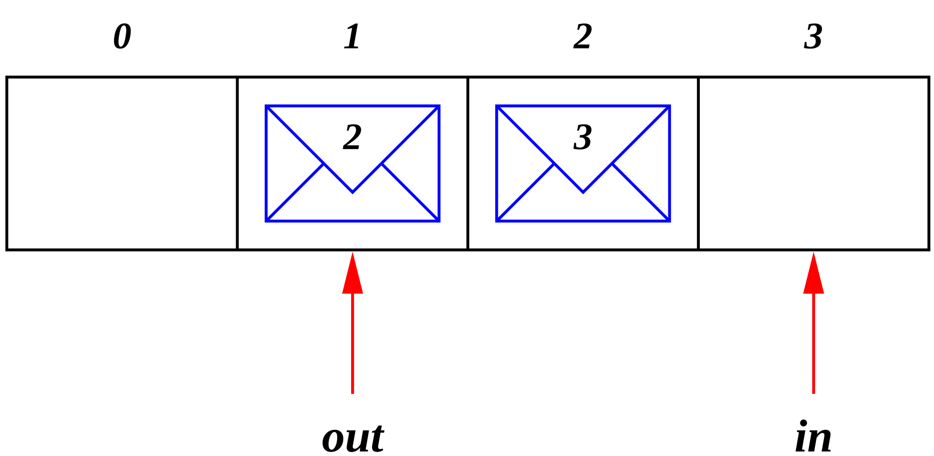
- *Length* liefert die Anzahl der in der Warteschlange befindlichen Elemente und *Full* liefert **TRUE**, falls die Warteschlange keine weiteren Elemente mehr aus Kapazitätsgründen aufnehmen kann, bevor nicht mindestens ein Element entfernt wird.

Warteschlangen auf Basis von Ringpuffern

Requests.om

```
TYPE
   QueueOfRequests =
      RECORD
         ring: ARRAY capacity OF Request;
         length: INTEGER; (* [0..capacity] *)
         in, out: INTEGER; (* [0..capacity-1] *)
      END;
```

- Warteschlangen mit fester Kapazität lassen sich vortrefflich mit Ringpuffern realisieren.
- Ringpuffer führen zwei Indizes: *in* verweist auf die nächste freie Position und *out* auf das nächste zu entfernende Element.
- Wenn *in* mit *out* identisch ist, läßt sich nur über *length* feststellen, ob der Ringpuffer völlig leer oder vollständig gefüllt ist.



Warteschlangen auf Basis von Ringpuffern

Requests.om

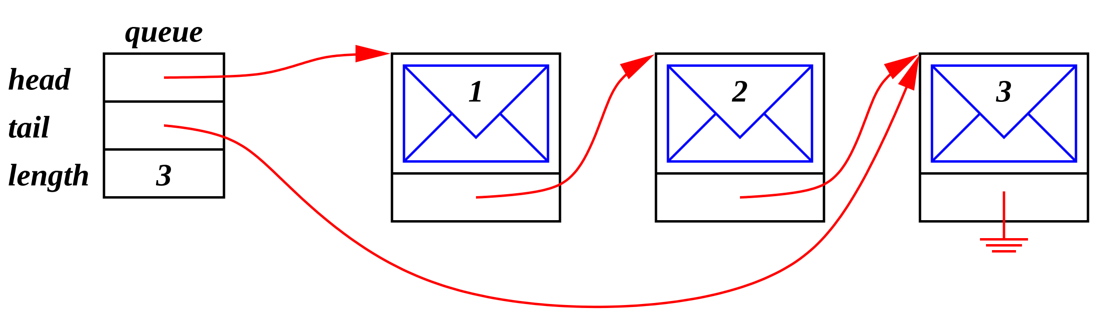
```
PROCEDURE InitQueue(VAR queue: QueueOfRequests);
BEGIN
   queue.length := 0; queue.in := 0; queue.out := 0;
END InitQueue;

PROCEDURE AddRequest(VAR queue: QueueOfRequests;
                     request: Request);
BEGIN
   ASSERT(queue.length < capacity);
   queue.ring[queue.in] := request;
   queue.in := (queue.in + 1) MOD capacity;
   INC(queue.length);
END AddRequest;

PROCEDURE RemoveRequest(VAR queue: QueueOfRequests;
                        VAR request: Request);
BEGIN
   ASSERT(queue.length > 0);
   request := queue.ring[queue.out];
   queue.out := (queue.out + 1) MOD capacity;
   DEC(queue.length);
END RemoveRequest;

PROCEDURE Length(queue: QueueOfRequests) : INTEGER;
BEGIN
   RETURN queue.length
END Length;

PROCEDURE Full(queue: QueueOfRequests) : BOOLEAN;
BEGIN
   RETURN queue.length = capacity
END Full;
```

Warteschlangen auf Basis von verzeigerten Elementen

Requests2.om

```
TYPE
   Element = POINTER TO ElementRec;
   ElementRec =
      RECORD
         request: Request;
         next: Element;
      END;

   QueueOfRequests =
      RECORD
         head, tail: Element;
         length: INTEGER;
      END;
```

- Warteschlangen lassen sich mit einfach verzeigerten Listen realisieren.
- Jedes Elemente zeigt dabei auf das nach ihm gekommene Element. Die Warteschlange selbst unterhält einen Zeiger an den Anfang (*head*) und das Ende (*tail*) der Liste.



Warteschlangen auf Basis von verzeigerten Elementen

Requests2.om

```
PROCEDURE InitQueue(VAR queue: QueueOfRequests);
BEGIN
   queue.length := 0; queue.head := NIL; queue.tail := NIL;
END InitQueue;

PROCEDURE AddRequest(VAR queue: QueueOfRequests;
                     request: Request);
   VAR
      element: Element;
BEGIN
   NEW(element); element.request := request;
   element.next := NIL;
   IF queue.head = NIL THEN
      queue.head := element;
   ELSE (* queue.tail # NIL *)
      queue.tail.next := element;
   END;
   queue.tail := element;
   INC(queue.length);
END AddRequest;
```

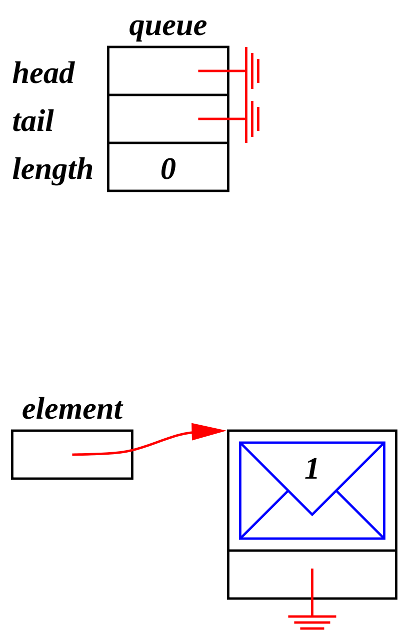
- Es gilt die Invariante: Entweder sind sowohl *queue.head* und *queue.tail* gleich **NIL** oder beide ungleich **NIL**.

Warteschlangen auf Basis von verzeigerten Elementen

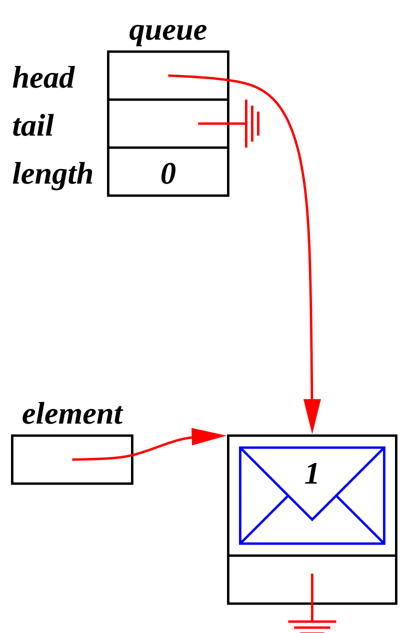
Requests2.om

```
PROCEDURE RemoveRequest(VAR queue: QueueOfRequests;
                        VAR request: Request);
BEGIN
   ASSERT(queue.length > 0);
   request := queue.head.request;
   queue.head := queue.head.next;
   IF queue.head = NIL THEN
      queue.tail := NIL;
   END;
   DEC(queue.length);
END RemoveRequest;

PROCEDURE Length(queue: QueueOfRequests) : INTEGER;
BEGIN
   RETURN queue.length
END Length;
```

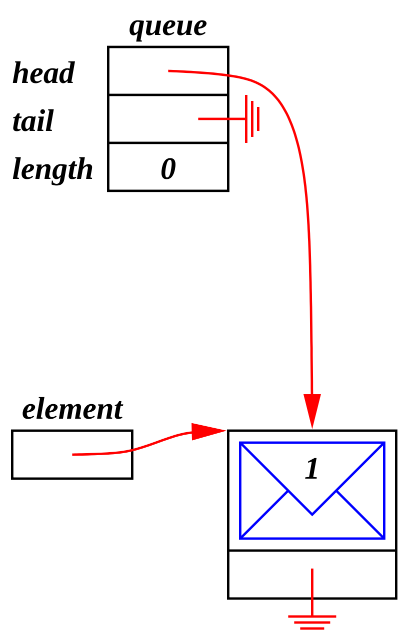
Einfügen in eine leere Warteschlange



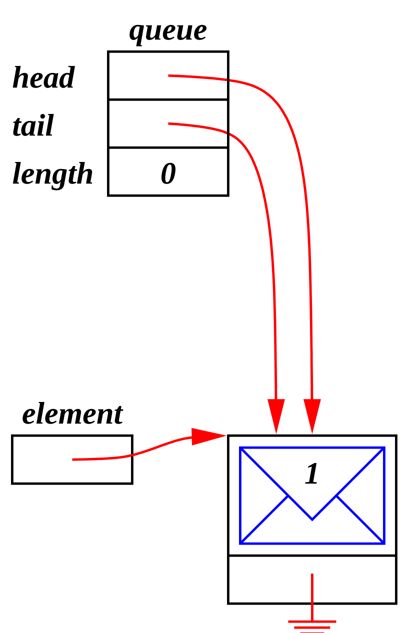
- `queue.head := element;`



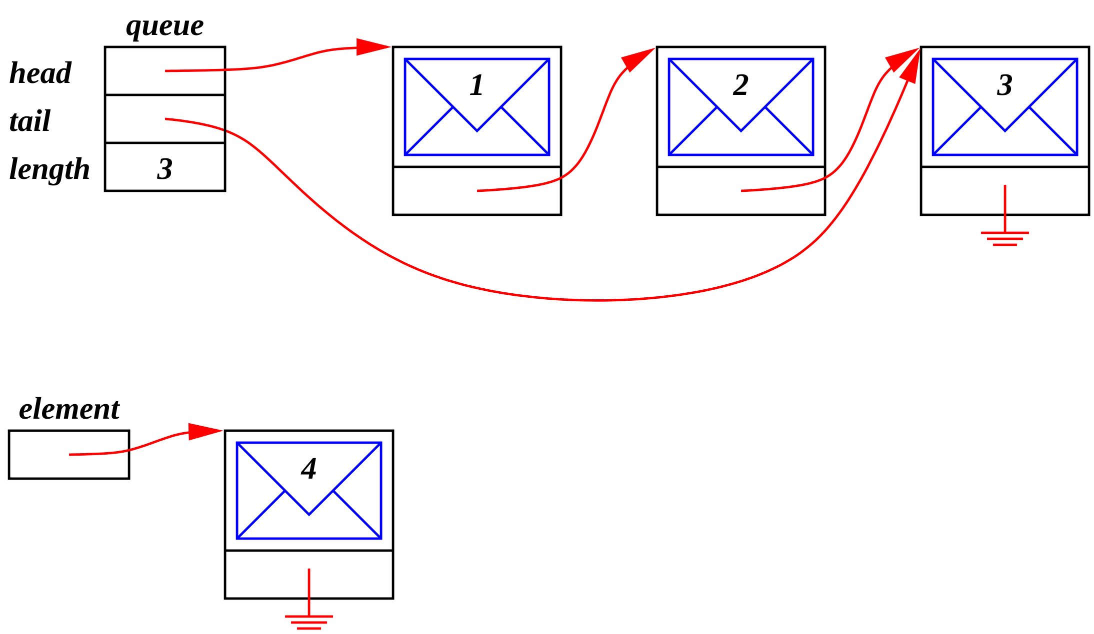
Einfügen in eine leere Warteschlange



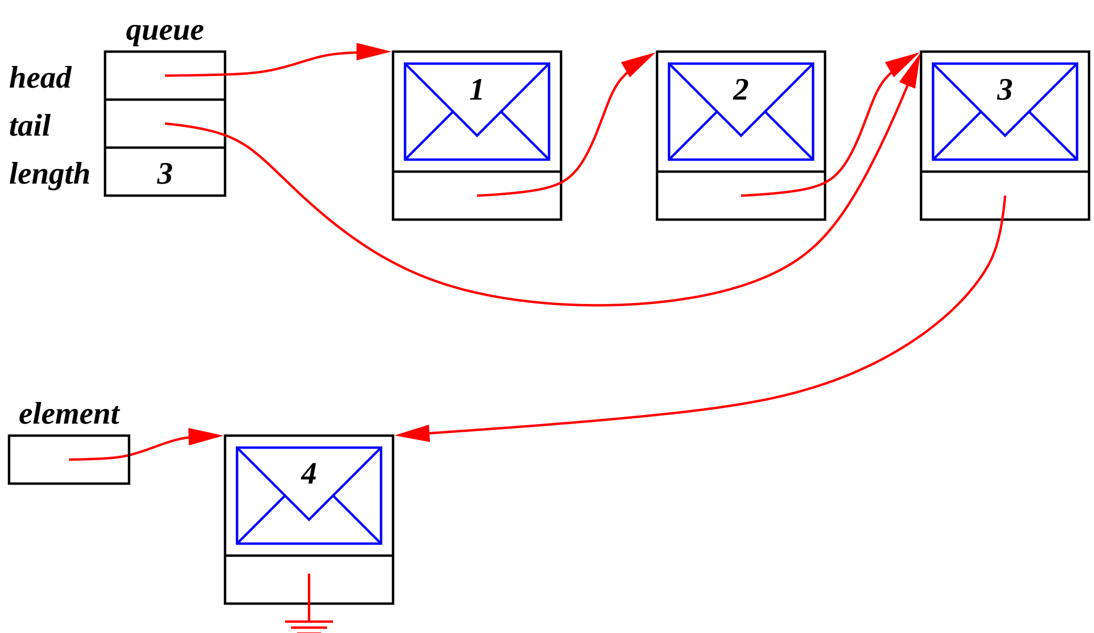
- `queue.tail := element;`



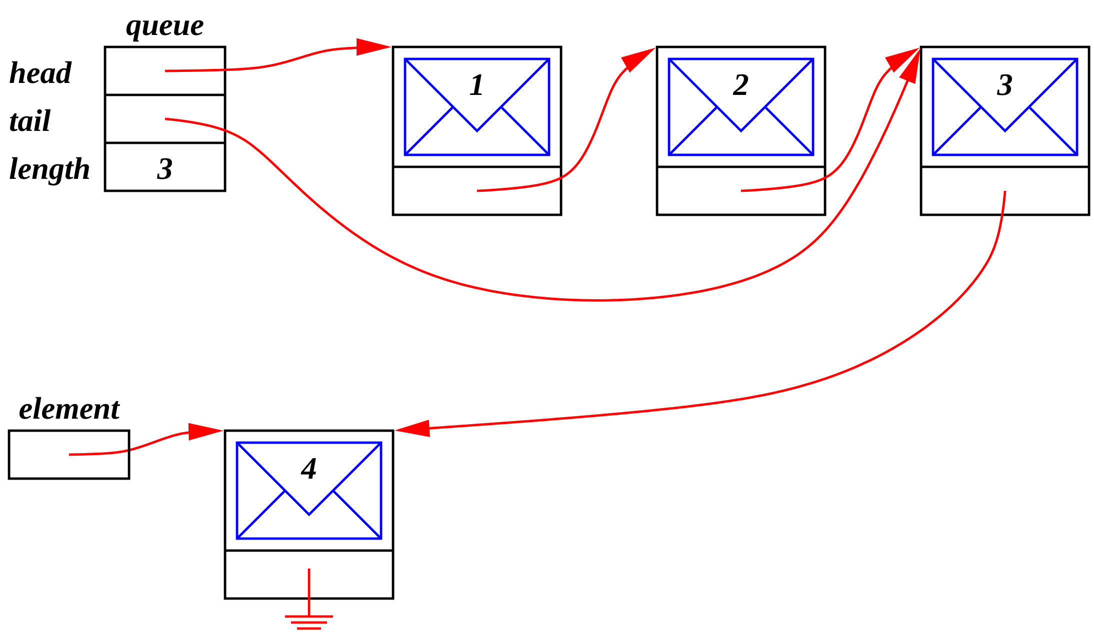
Einfügen in eine gefüllte Warteschlange



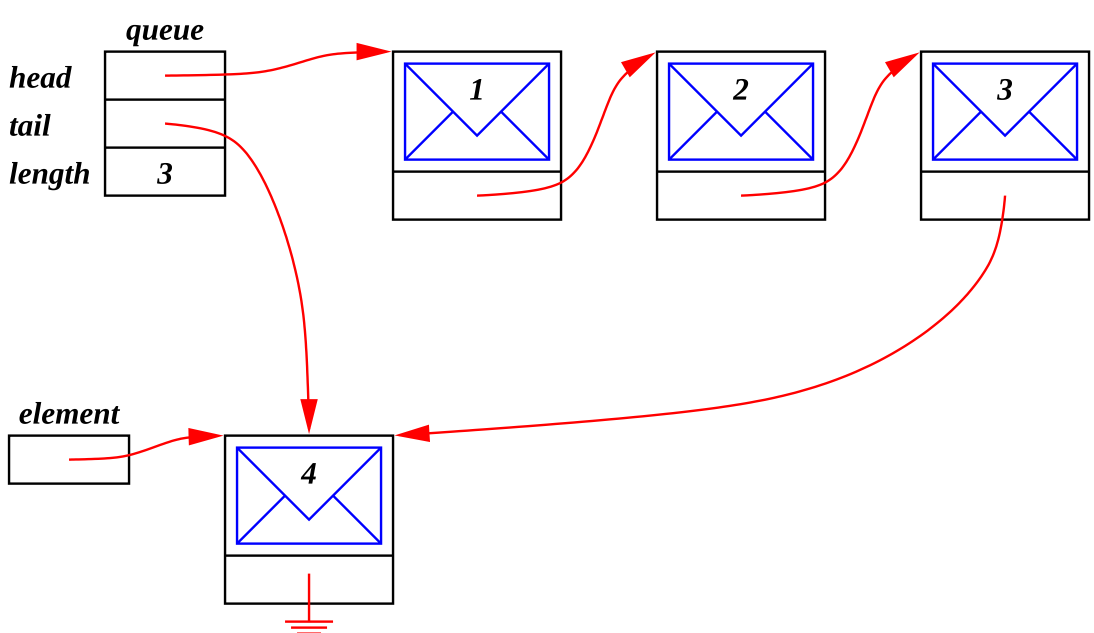
- `queue.tail.next := element;`



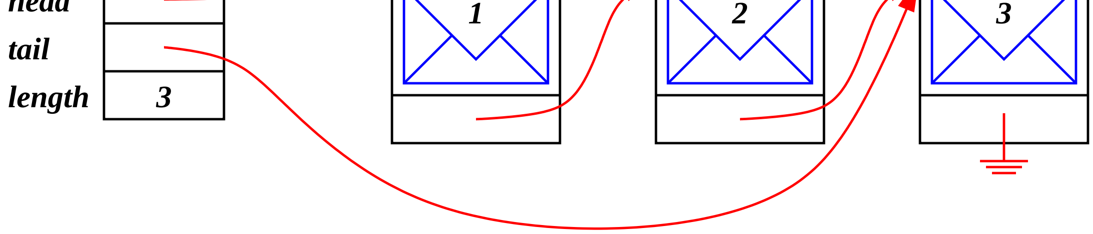
Einfügen in eine gefüllte Warteschlange



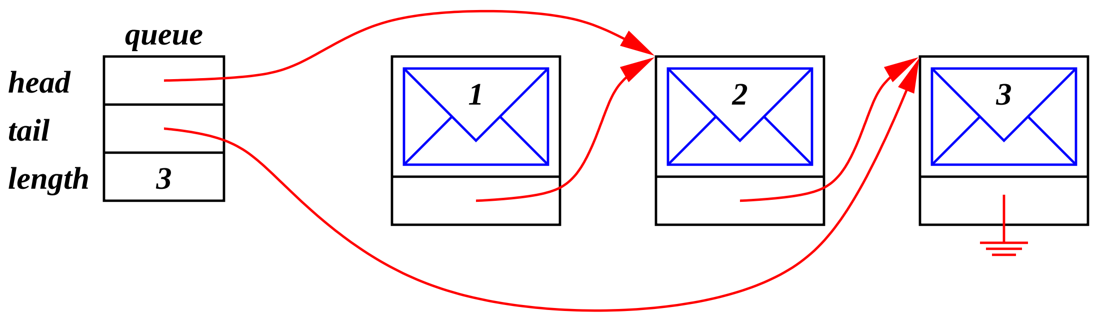
- `queue.tail := element;`



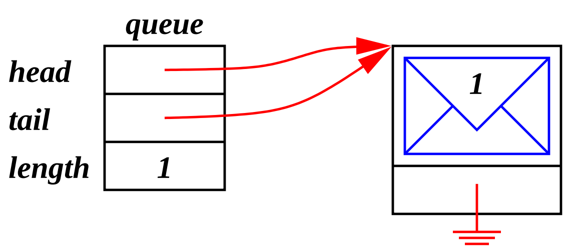
Entfernen aus einer Warteschlange



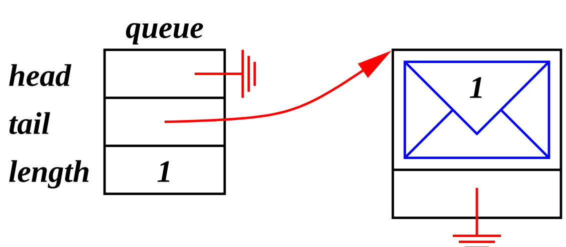
- `queue.head := queue.head.next;`



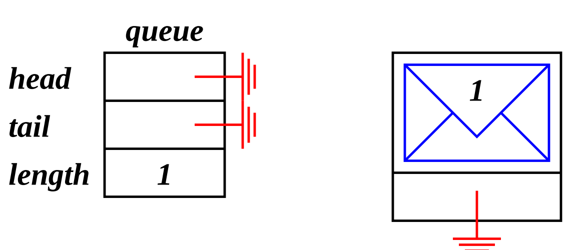
Entfernen des letzten Elements einer Warteschlange



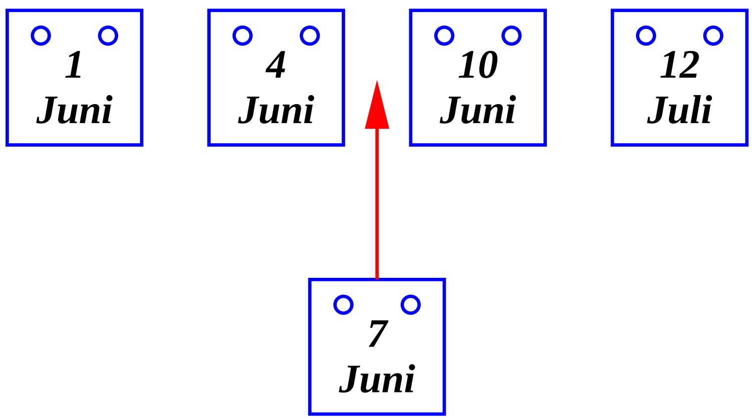
- `queue.head := queue.head.next;`



- `queue.tail := NIL;`



Sortierte Listen



- Bei sortierten Listen besitzt jedes Element einen Schlüssel, für den es eine vollständige Ordnungs-Relation $\leq$ gibt.

- Eine Ordnungs-Relation $\leq$ ist vollständig, wenn

  - $a \leq a$ (Reflexivität)

  - $a \leq b \wedge b \leq a \Rightarrow a = b$ (Antisymmetrie)

  - $a \leq b \wedge b \leq c \Rightarrow a \leq c$ (Transitivität)

Operationen für sortierte Listen

```
PROCEDURE InitSortedList(VAR list: SortedList);
PROCEDURE InsertElement(VAR list: SortedList;
                        element: Element);
PROCEDURE RemoveFirstElement(VAR list: SortedList);
PROCEDURE GetFirstElement(list: SortedList;
                          VAR element: Element);
PROCEDURE IterateSortedList(VAR list: SortedList);
PROCEDURE GetElement(VAR list: SortedList;
                     VAR element: Element) : BOOLEAN;
PROCEDURE Length(list: SortedList) : INTEGER;
```

- *InsertElement* fügt *element* in *list* so ein, daß alle Elemente weiterhin in der Liste sortiert bleiben.

- *RemoveFirstElement* entfernt das erste Element der Liste, das wegen der Sortierung den niedrigsten Schlüsselwert besitzt.

- *GetFirstElement* liefert das erste Element aus der Liste (ohne es zu entfernen).

- Mit *IterateSortedList* und *GetElement* ist es möglich, sich alle Elemente der Liste in sortierter Reihenfolge anzusehen.

- Weitere Operationen sind denkbar, z.B. schlüsselbasiertes Suchen und Entfernen.

Sortierte Listen auf Basis von verzeigerten Elementen

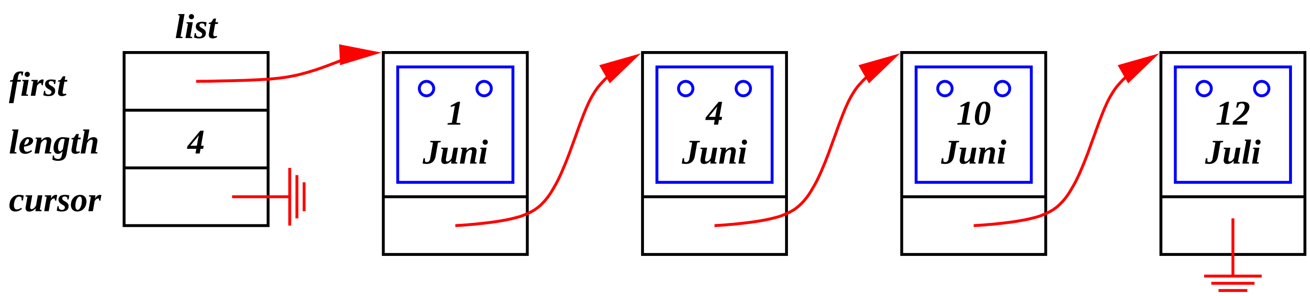
CalendarMan.om

```
TYPE
   Appointment =
      RECORD
         time: Times.Time; (* time of appointment *)
         text: ARRAY 128 OF CHAR; (* description *)
      END;
   Element = POINTER TO ElementRec;
   ElementRec =
      RECORD
         appointment: Appointment;
         next: Element; (* next in ascending order *)
      END;
   ListOfAppointments =
      RECORD
         first: Element; (* sorted by time in
                            ascending order *)
         length: INTEGER; (* number of appointments *)
         cursor: Element; (* next element to be returned *)
      END;
```

- Für kleinere Mengen an Elementen bieten sich aus Gründen der Einfachheit verzeigerte Listen an.

- *first* zeigt auf das erste Element, *cursor* dient zum Durchlaufen der Liste mit *IterateSortedList* und *GetElement*.

Sortierte Listen auf Basis von verzeigerten Elementen



CalendarMan.om

```
PROCEDURE InitListOfAppointments
            (VAR list: ListOfAppointments);
BEGIN
   list.first := NIL; list.length := 0; list.cursor := NIL;
END InitListOfAppointments;

PROCEDURE GetFirstAppointment(list: ListOfAppointments;
                              VAR appointment: Appointment);
BEGIN
   ASSERT(list.length > 0);
   appointment := list.first.appointment;
END GetFirstAppointment;

PROCEDURE NumberOfAppointments
            (list: ListOfAppointments) : INTEGER;
BEGIN
   RETURN list.length
END NumberOfAppointments;
```

Sortierte Listen auf Basis von verzeigerten Elementen

CalendarMan.om

```
PROCEDURE InsertAppointment(VAR list: ListOfAppointments;
                            appointment: Appointment);
   VAR
      ptr, prev, element: Element;
BEGIN
   NEW(element); element.appointment := appointment;
   IF (list.first = NIL) OR
         (Op.Compare(appointment.time,
            list.first.appointment.time) < 0) THEN
      (* insert element at the beginning of the list *)
      element.next := list.first; list.first := element;
   ELSE
      prev := list.first; ptr := list.first.next;
      WHILE (ptr # NIL) &
            (Op.Compare(appointment.time,
               ptr.appointment.time) >= 0) DO
         prev := ptr; ptr := ptr.next;
      END;
      (* following holds now:
            (ptr = prev.next) &
            (appointment.time >= prev.appointment.time) &
            ((ptr = NIL) OR
               (appointment.time < ptr.appointment.time))
         we insert element between prev and ptr:
      *)
      element.next := ptr; prev.next := element;
   END;
   INC(list.length);
END InsertAppointment;
```

Sortierte Listen auf Basis von verzeigerten Elementen

CalendarMan.om

```
PROCEDURE RemoveFirstAppointment
            (VAR list: ListOfAppointments);
BEGIN
   ASSERT(list.length > 0);
   list.first := list.first.next;
   DEC(list.length);
END RemoveFirstAppointment;

PROCEDURE GetFirstAppointment(list: ListOfAppointments;
                     VAR appointment: Appointment);
BEGIN
   ASSERT(list.length > 0);
   appointment := list.first.appointment;
END GetFirstAppointment;

PROCEDURE IterateListOfAppointments
            (VAR list: ListOfAppointments);
BEGIN
   list.cursor := list.first;
END IterateListOfAppointments;

PROCEDURE GetAppointment(VAR list: ListOfAppointments;
                VAR appointment: Appointment) : BOOLEAN;
BEGIN
   IF list.cursor = NIL THEN
      RETURN FALSE
   ELSE
      appointment := list.cursor.appointment;
      list.cursor := list.cursor.next;
      RETURN TRUE
   END;
END GetAppointment;
```

Exkurs: Zeit in der Ulmer Oberon-Bibliothek

PrintDate.om

```
MODULE PrintDate;

   IMPORT Clocks, Dates, Print, Times;

   VAR
      now: Times.Time;

   PROCEDURE PrintTime(time: Times.Time);
      VAR
         date: Dates.LongInfoRec;
   BEGIN
      Dates.GetLong(time, date);
      Print.F7("%02d.%02d.%02d %02d:%02d:%02d %s\n",
         date.day, date.month, date.year,
         date.hour, date.minute, date.second,
         date.tzInfo.name);
   END PrintTime;

BEGIN
   Clocks.GetTime(Clocks.system, now);
   PrintTime(now);
END PrintDate.
```

- *Times.Time* ist ein abstrakter Datentyp, der mit beliebigen Repräsentierungen der Zeit (sowohl absolut als auch relativ) arbeiten kann.

- *Clocks* ist eine Abstraktion für beliebige Uhren; *Clocks.system* ist die lokale Systemuhr.

Exkurs: Zeit in der Ulmer Oberon-Bibliothek

PrintDate.om

```
PROCEDURE PrintTime(time: Times.Time);
   VAR
      date: Dates.LongInfoRec;
BEGIN
   Dates.GetLong(time, date);
   Print.F7("%02d.%02d.%02d %02d:%02d:%02d %s\n",
      date.day, date.month, date.year,
      date.hour, date.minute, date.second,
      date.tzInfo.name);
END PrintTime;
```

- Das Modul *Dates* kann Zeitangaben entsprechend dem gängigen Kalender konvertieren.

- Das Modul *Print* funktioniert ähnlich wie *printf* in der C-Bibliothek. Hierbei spezifiziert ein Format wie die anschließenden Parameter in den auszugebenden Text eingefügt werden. Dabei operiert jede %-Sequenz als Platzhalter für einen der auszugebenden Parameter.

- `%02d` steht für dezimal, 2 Ziffern mit führenden Nullen.

- `%s` steht für eine Zeichenkette.

Exkurs: Zeit in der Ulmer Oberon-Bibliothek

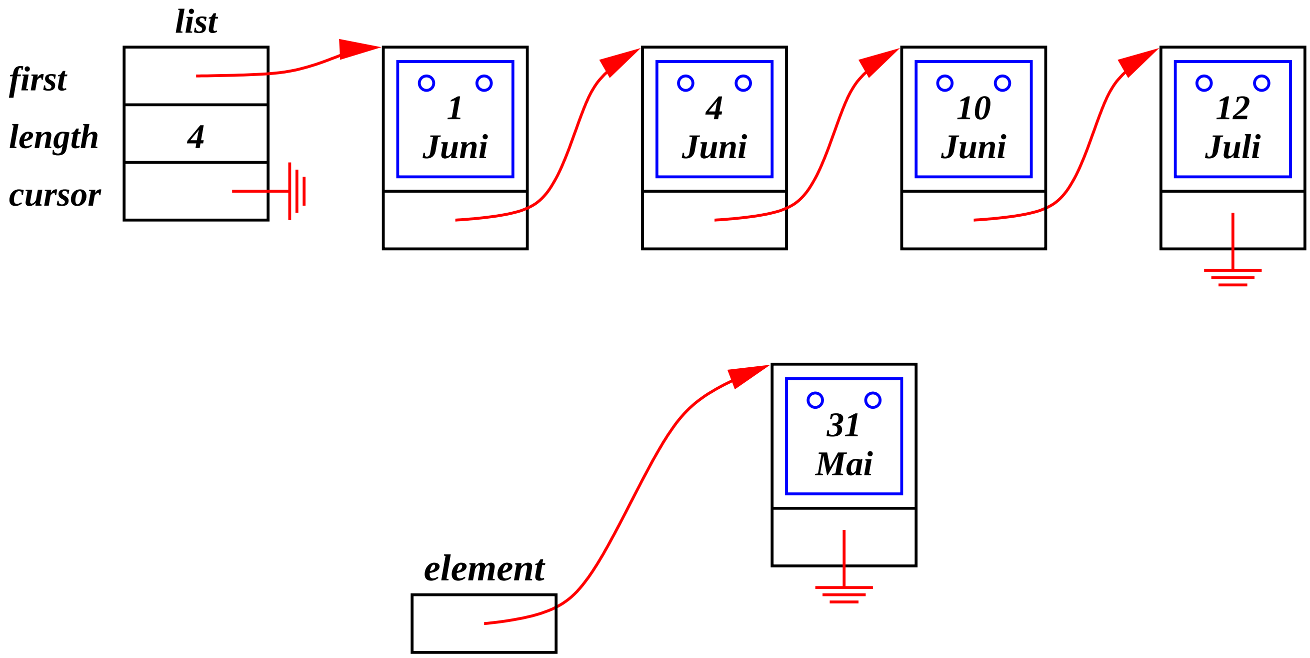
TimeDiff.om

```
PROCEDURE PrintTimeDiff(time1, time2: Times.Time);
   VAR
      diff: Times.Time;
      dateval: Dates.ValueRec;
BEGIN
   (* avoid differences with a negative result *)
   IF Op.Compare(time1, time2) >= 0 THEN
      (* diff := time1 - time2 *)
      Op.Sub3(diff, time1, time2);
   ELSE
      Op.Sub3(diff, time2, time1);
   END;
   Dates.GetValue(diff, dateval);
   Write.Int(dateval.days, 1); Write.String(" days, ");
   Write.Int(dateval.hours, 1); Write.String(" hours, and ");
   Write.Int(dateval.minutes, 1); Write.Line(" minutes.");
END PrintTimeDiff;
```

- Operationen mit Zeiten sind möglich, so kann z.B. aus der Differenz zweier absoluter Zeiten eine relative Zeitangabe gebildet werden.

- *Op* wird hier als Abkürzung für das Modul *Operations* verwendet, das arithmetische Operationen für beliebige Datentypen (wie z.B. *Times.Time*) unterstützen kann.

Einfügen in eine sortierte Liste

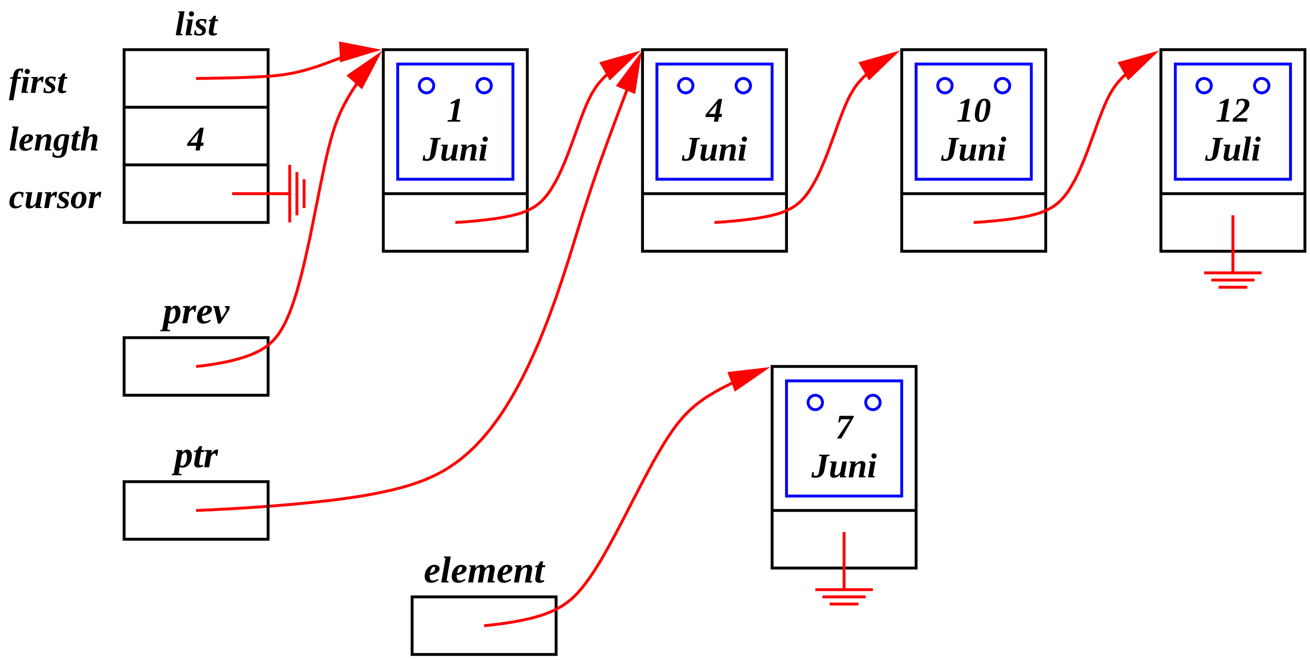


- Wenn *list.first* gleich **NIL** oder *element.appointment.time* kleiner als *list.first.appointment.time* ist, dann ist das neue Element ganz zu Beginn der Liste einzufügen.

CalendarMan.om

```
IF (list.first = NIL) OR
      (Op.Compare(appointment.time,
         list.first.appointment.time) < 0) THEN
   (* insert element at the beginning of the list *)
   element.next := list.first; list.first := element;
ELSE
   (* ... *)
END;
```

Einfügen in eine sortierte Liste



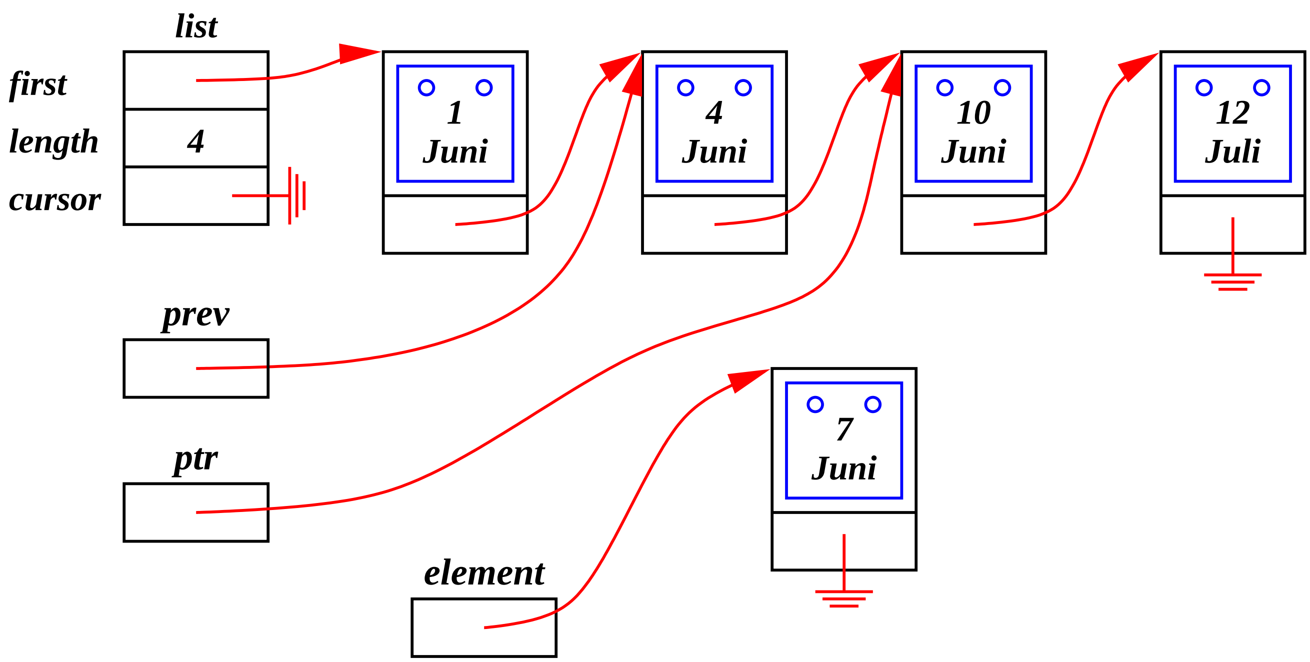
- Wenn *list.first* ungleich **NIL** und *appointment.time* größer oder gleich *list.first.appointment.time* ist, dann ist *element* in jedem Falle hinter *list.first* einzufügen.

- Folgende **WHILE**-Schleife sucht dann die Einfügeposition, wobei *element* immer irgendwo hinter *prev* einzufügen ist.

CalendarMan.om

```
prev := list.first; ptr := list.first.next;
WHILE (ptr # NIL) &
      (Op.Compare(appointment.time,
         ptr.appointment.time) >= 0) DO
   prev := ptr; ptr := ptr.next;
END;
```

Einfügen in eine sortierte Liste

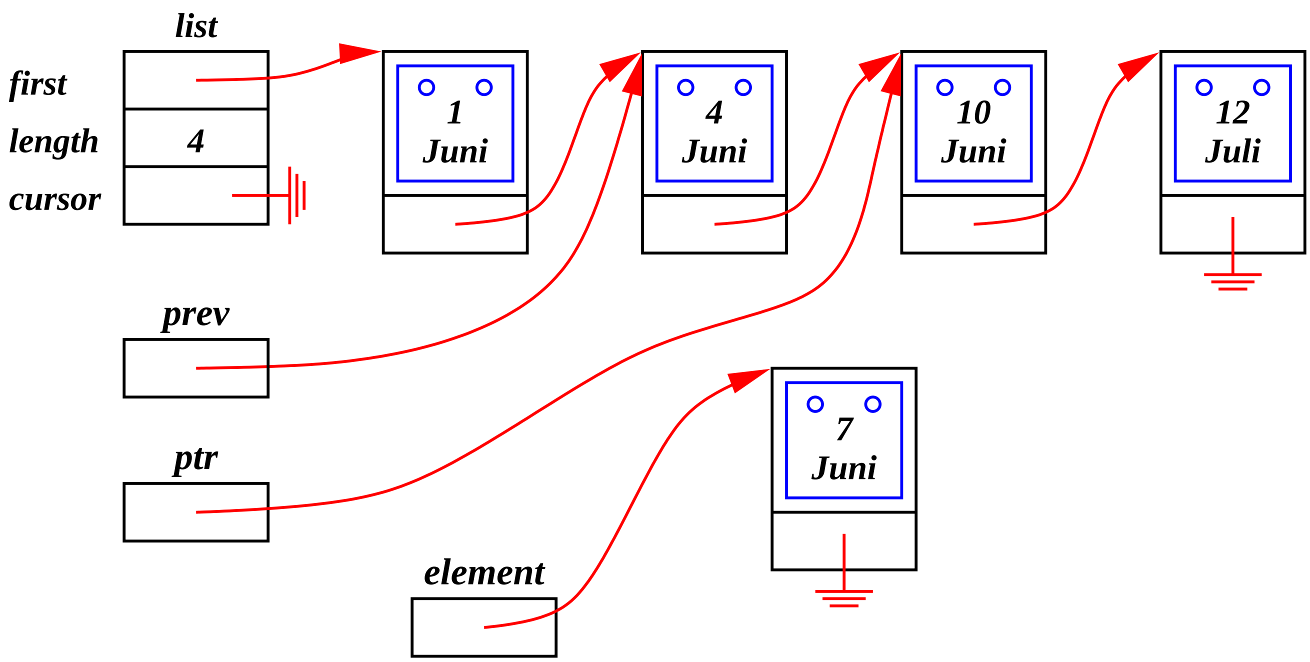


- Nach der Schleife, die die Einfügeposition sucht, gilt:

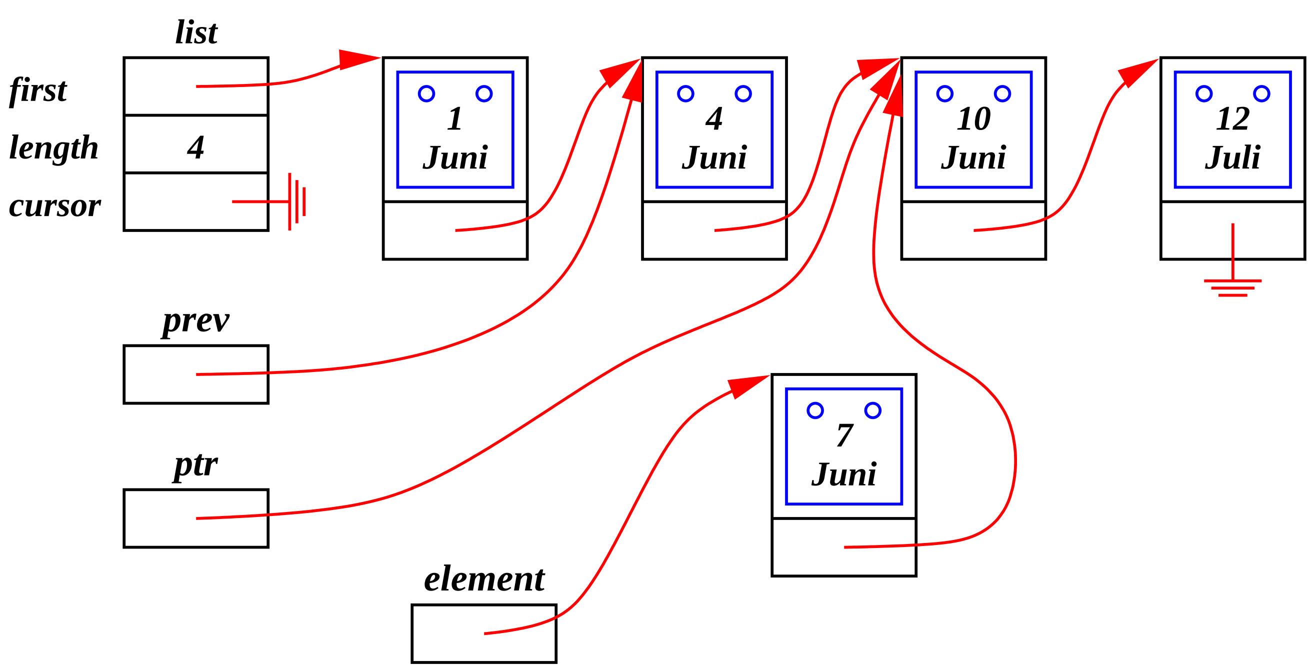
  - `ptr = prev.next`
  - `Op.Compare(appointment.time, prev.appointment.time) >= 0`
  - `(ptr = NIL) OR`
    `Op.Compare(appointment.time, ptr.appointment.time) < 0)`

- Entsprechend wird das neue Element zwischen *prev* und *ptr* eingefügt:
- `element.next := ptr; prev.next := element;`

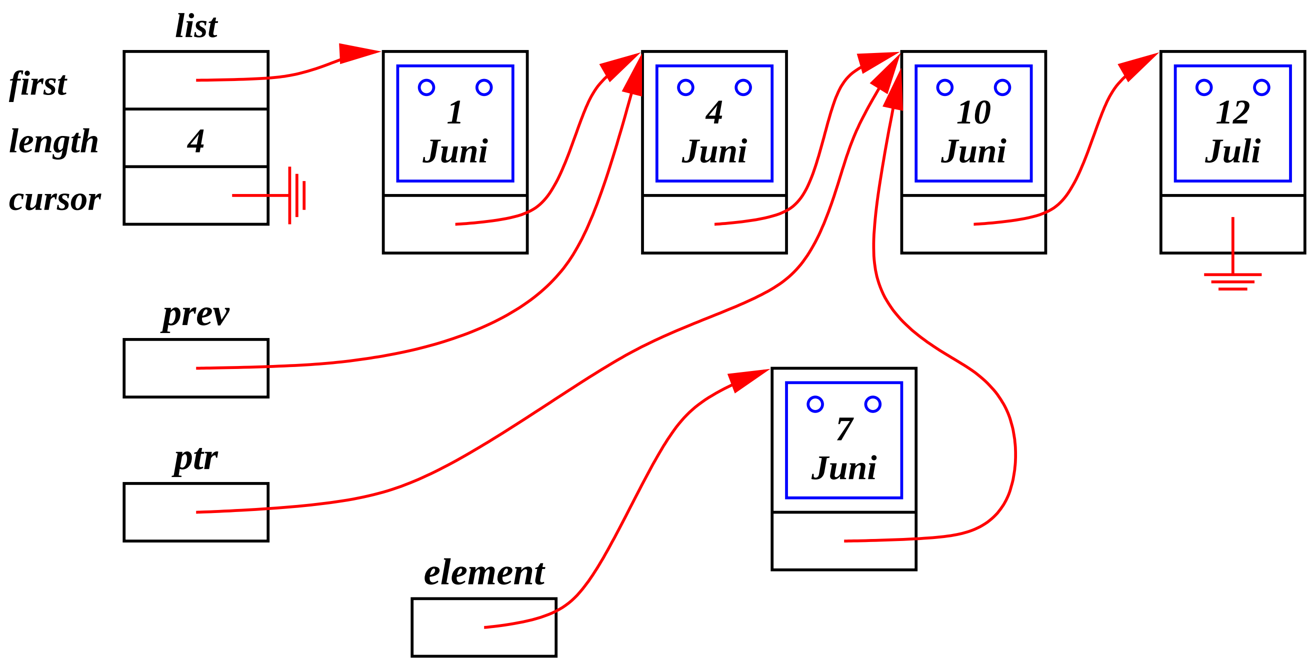
Einfügen in eine sortierte Liste



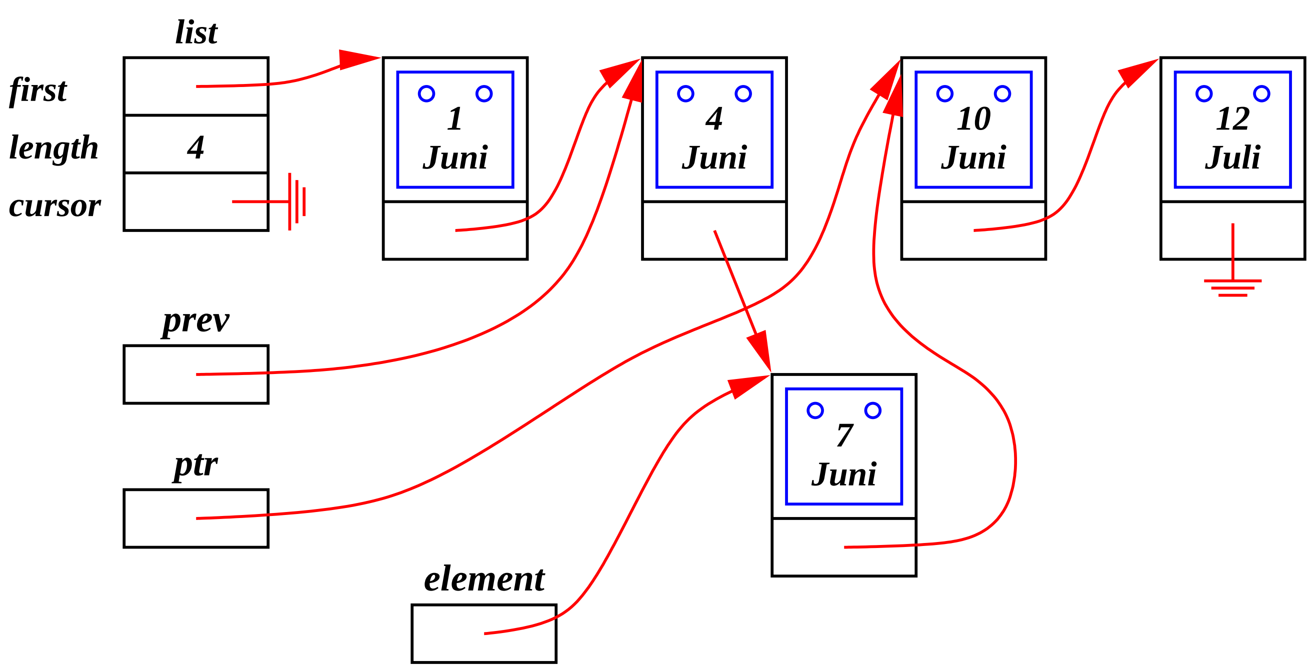
- `element.next := ptr;`



Einfügen in eine sortierte Liste



- `prev.next := element;`

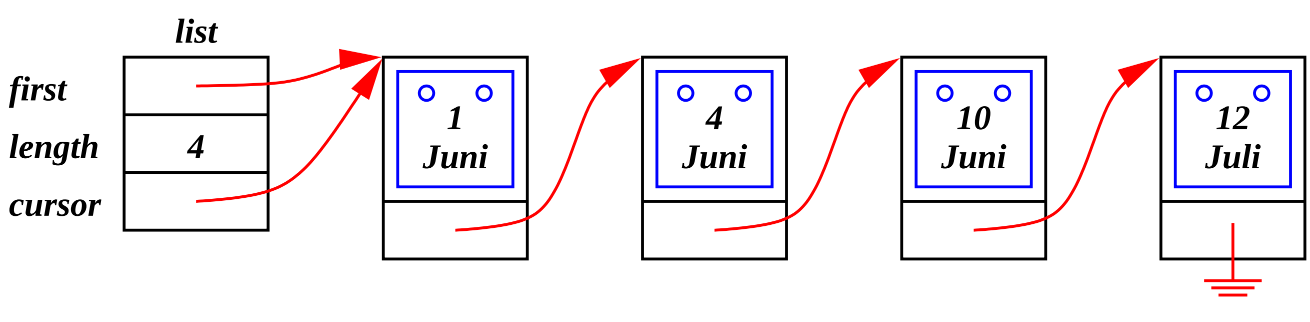


Das Durchlaufen einer Liste

CalendarMan.om

```
IterateListOfAppointments(list);
WHILE GetAppointment(list, appointment) DO
   WriteAppointment(Streams.stdout, appointment);
END;
```

- Mit *IterateListOfAppointments* wird *list.cursor* auf das 1. Element gesetzt.
- *GetAppointment* liefert jeweils den Termin zurück, auf den *list.cursor* zeigt, und setzt anschließend *list.cursor* um eines weiter.
- Anfangssituation unmittelbar nach dem Aufruf von *IterateListOfAppointments*:

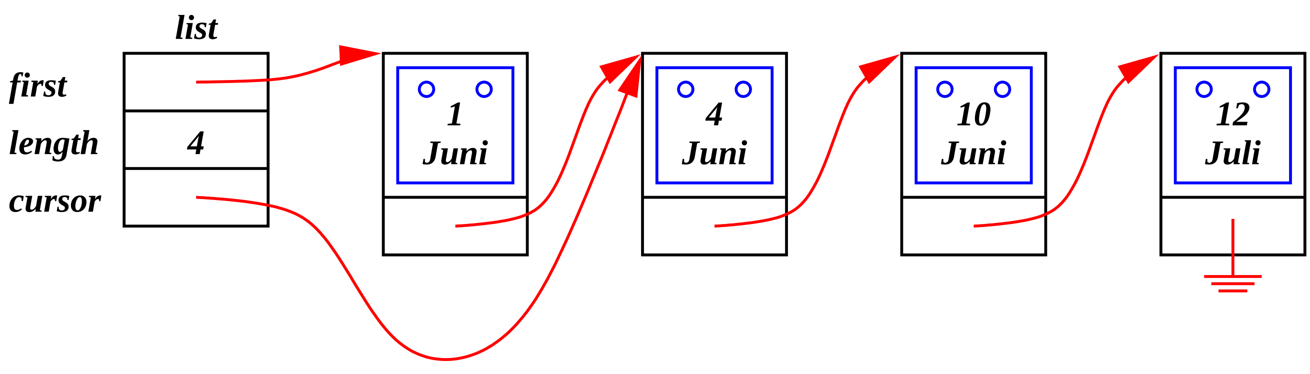


Das Durchlaufen einer Liste

CalendarMan.om

```
PROCEDURE GetAppointment(VAR list: ListOfAppointments;
                         VAR appointment: Appointment) : BOOLEAN;
BEGIN
   IF list.cursor = NIL THEN
      RETURN FALSE
   ELSE
      appointment := list.cursor.appointment;
      list.cursor := list.cursor.next;
      RETURN TRUE
   END;
END GetAppointment;
```

- Situation nach dem 1. Aufruf von *GetAppointment*:



Literaturhinweise zu Zeigern und Listen

- Martin Reiser et al., “Programming Oberon”
  zeigt in Kapitel 9 “Dynamic data structures and pointer types” den Umgang in Oberon mit Zeigern und stellt einige einfache dynamische Datenstrukturen einschließlich linearer Listen vor.

- Donald E. Knuth, “The Art of Computer Programming”, Volume 1
  Der Abschnitt 2.2 geht auf lineare Listen ein und berücksichtigt dabei insbesondere Allokationsmöglichkeiten auf maschinennaher Ebene.

- Alfred V. Aho et al., “Data Structures and Algorithms”
  Kapitel 2 führt Stapel und Warteschlangen mit Beispielen in Pascal ein.

- Thomas E. Cormen, Charles E. Leiserson, Ronald L. Rivest, “Introduction to Algorithms”
  Kapitel 11 geht auf verzeigerte Listen ein und bietet auch Hinweise, wie Zeiger und verzeigerte Elemente auf Basis von ARRAYs realisiert werden können.

Modularisierung und Abstraktionen

- Die Verwaltung von Listen und anderen Datenstrukturen wird immer wieder benötigt. Daher ist es erstrebenswert, wenn

  - die Verwaltung solcher Datenstrukturen in separate Module ausgelagert werden und

  - diese beliebige Elementtypen akzeptieren würden.

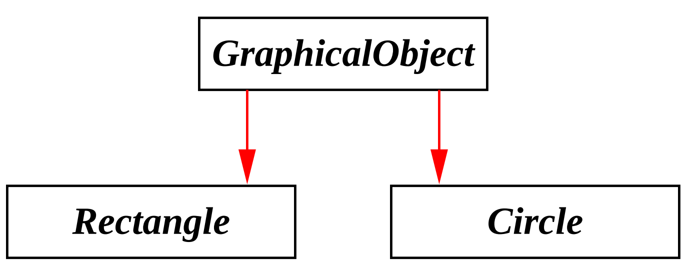
- Um letzteres zu ermöglichen bzw. zu erleichtern, gibt es in Oberon das Konzept der Typ-Erweiterungen.

Typ-Erweiterungen

```
TYPE
   GraphicalObject =
      RECORD
         x, y: INTEGER; (* center point *)
      END;
   Rectangle =
      RECORD
         (GraphicalObject)
         width, height: INTEGER;
      END;
   Circle =
      RECORD
         (GraphicalObject)
         radius: INTEGER;
      END;
```

- Record-Typen können als Erweiterung eines anderen Record-Typs definiert werden. In diesem Falle wird der zu erweiternde Record-Typ in Klammern unmittelbar hinter dem Schlüsselwort **RECORD** angegeben.

- Ein Record-Typ kann nur die Erweiterung eines anderen Record-Typs sein – nicht von mehreren. Somit ergibt sich die Struktur eines Baumes bzw. die mehrerer Bäume.



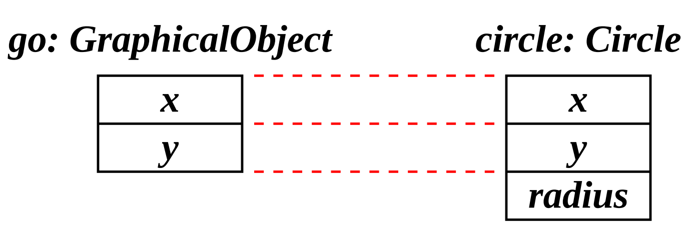
Typ-Erweiterungen

```
PROCEDURE InitCircle(VAR circle: Circle;
                     x, y, radius: INTEGER);
BEGIN
   circle.x := x; circle.y := y;
   circle.radius := radius;
END InitCircle;
```

- Wenn ein Record eine Erweiterung eines anderen Records ist, stehen alle Komponenten zur Verfügung.

- Der wesentliche Vorteil der Typ-Erweiterungen besteht in der damit verbundenen großzügigen Kompatibilität. So kann ein erweiterter Record (wie *Circle*) einem Record des zugehörigen Basistyps (wie z.B. *GraphicalObject*) zugewiesen werden. Dabei findet eine Projektion statt, d.h. die erweiternden Komponenten werden nicht kopiert.

```
VAR go: GraphicalObject; circle: Circle; rect: Rectangle;
(* ... *)
go := circle; (* OK *)
circle := go; (* not permitted *)
circle := rect; (* not permitted *)
```



Typ-Erweiterungen

```
PROCEDURE Move(VAR go: GraphicalObject; newx, newy: INTEGER);
BEGIN
   go.x := newx;
   go.y := newy;
END Move;
```

- *Move* kann für eine beliebige Erweiterung von *GraphicalObject* aufgerufen werden; so ist z.B.
  `Move(circle, 0, 0);`
  zulässig.

- Bei **VAR**-Parametern findet eine Projektion nicht statt.

Erweiterungen und Basistypen

Gegeben sei eine Deklaration der Form

```
T0 = RECORD (T) ... END;
```

Dann ist *T0* eine **direkte Erweiterung** von *T*, und *T* ist der **direkte Basistyp** von *T0*.

Erweiterte Typen können erneut erweitert werden: Ein Record-Typ *U* ist eine **Erweiterung** des Record-Typs *T* genau dann, falls

- $U = T$ oder

- *U* eine direkte Erweiterung von *T* ist oder

- *U* eine direkte Erweiterung einer Erweiterung von *T* ist.

Entsprechend ist ein Record-Typ *T* genau dann ein **Basistyp** eines Record-Typs *U*, wenn *U* eine Erweiterung von *T* ist.

Typ-Erweiterungen für Zeiger

```
TYPE
   GraphicalObject = POINTER TO GraphicalObjectRec;
   GraphicalObjectRec =
      RECORD
         x, y: INTEGER; (* center point *)
      END;
   Rectangle = POINTER TO RectangleRec;
   RectangleRec =
      RECORD
         (GraphicalObjectRec)
         width, height: INTEGER;
      END;
   Circle = POINTER TO CircleRec;
   CircleRec =
      RECORD
         (GraphicalObjectRec)
         radius: INTEGER;
      END;
```

- Die Erweiterungsbeziehung und die damit verbundene erweiterte Kompatibilität erstreckt sich auch auf die zugehörigen Zeigertypen.

- Seien folgende Definitionen gegeben:

  ```
  TYPE P = POINTER TO T;
  TYPE Q = POINTER TO U;
  ```

  Dann ist ein Zeigertyp Q genau dann eine Erweiterung eines Zeigertyps P, wenn U eine Erweiterung von T ist. Ein Zeigertyp P ist genau dann ein Basistyp eines Zeigertyps Q, wenn Q eine Erweiterung von P ist.

Laufzeitfehler bei VAR-Parametern

```
PROCEDURE Copy(VAR source, dest: GraphicalObject);
BEGIN
   dest := source;
END Copy;

(* ... *)

Copy(circle, rectangle);
```

- Jedes der beiden Teile in diesem Beispiel ist – für sich genommen – vollkommen korrekt und wird vom Compiler entsprechend auch akzeptiert.

- Wenn zur Laufzeit jedoch eine Zuweisung eines Kreises an einen Rechteck probiert wird, führt dies zu einem Laufzeitfehler.

- Bei **VAR**-Parametern wird also nicht nur der **angegebene Typ** des Parameters zur Übersetzzeit betrachtet, sondern auch der **tatsächliche Typ** zur Laufzeit einer Typ-Überprüfung bei einer Zuweisung oder Parameterübergabe unterworfen.

- Dies gilt sowohl bei Record- als auch bei Zeiger-Typen.

Heterogene Listen

```
TYPE
   Element = POINTER TO ElementRec;
   ElementRec =
      RECORD
         go: GraphicalObject;
         next: Element;
      END;
   ListOfGraphicalObjects =
      RECORD
         head, tail: Element;
         length: INTEGER;
      END;

PROCEDURE AddElement(list: ListOfGraphicalObjects;
                     go: GraphicalObject);
   VAR element: Element;
BEGIN
   NEW(element); element.go := go; element.next := NIL;
   IF list.head = NIL THEN
      list.head := element;
   ELSE
      list.tail.next := element;
   END;
   list.tail := element; INC(list.length);
END AddElement;

(* ... *)
AddElement(list, circle); AddElement(list, rectangle);
```

- Listen akzeptieren als Element alle Erweiterungen des vorgegebenen Basistyps.

- Wenn *GraphicalObject* ein Record-Typ ist, geht hingegen (durch die Projektion) die Erweiterung verloren – bei Zeigern gibt es jedoch keinen Informationsverlust.

Typen-Tests

- Wenn ein Zeiger oder Variablenparameter gegeben ist, dann sind zwei Typen damit assoziiert: der **statisch bekannte Typ** und der **dynamische Typ** (d.h. der zur Laufzeit bestimmte Typ).

- Dabei gilt, daß der dynamische Typ eine Erweiterung des statischen Typs sein muß.

- Oberon bietet den **IS**-Operator an, der es erlaubt, den dynamischen Typ zu überprüfen:

```
PROCEDURE Draw(VAR go: GraphicalObject);
BEGIN
   IF go IS Circle THEN
      (* draw a circle *)
   ELSIF go IS Rectangle THEN
      (* draw a rectangle *)
   ELSE
      (* unknown *)
   END;
END Draw;
```

- *variable* **IS** *Type*
  liefert dabei genau dann **TRUE**, falls der dynamische Typ von *variable* eine Erweiterung von *Type* ist.

- Der Compiler läßt Typen-Tests nicht zu, wenn *Type* keine Erweiterung des statischen Typs von *variable* ist.

Typ-Zusicherungen

- Eine Typzusicherung sichert zu, daß der dynamische Typ einer Variablen eine Erweiterung eines gegebenen Typs ist.
- Wenn die Typzusicherung zur Laufzeit nicht erfüllt ist, gibt es einen Laufzeitfehler.
- Ist die Typzusicherung erfolgreich, so läßt sich die Variable so behandeln, als würde ihr statischer Typ dem angegebenen Typ entsprechen.
- Dies ermöglicht insbesondere den Zugang zu den Komponenten der entsprechenden Erweiterung:

```
(* draw a circle *)
DrawCircle(go(Circle));

(* access radius component of go *)
diameter := go(Circle).radius * 2;
```

Regionale Typ-Zusicherungen

```
PROCEDURE Area(go: GraphicalObject) : REAL;
BEGIN
   IF go IS Rectangle THEN
      WITH go: Rectangle DO
         RETURN go.width * go.height
      END;
   ELSIF go IS Circle THEN
      WITH go: Circle DO
         RETURN Math.pi * go.radius * go.radius
      END;
   (* ... *)
   END;
END Area;
```

- Einzelne Typ-Zusicherungen blähen den Programmtext zu sehr auf, wenn sie mehrfach benötigt werden.

- Mit der **WITH**-Anweisung erstreckt sich eine Typ-Zusicherung über alle eingeschlossenen Anweisungen.

Modulkonzept in Oberon

Greetings.od

```
DEFINITION Greetings;

   IMPORT Streams;

   PROCEDURE Hello(s: Streams.Stream);

END Greetings.
```

Greetings.om

```
MODULE Greetings;

   IMPORT Streams, Write;

   PROCEDURE Hello(s: Streams.Stream);
   BEGIN
      Write.LineS(s, "Hello!");
   END Hello;

END Greetings.
```

- Ein Modul kann eine in sich konsistente beliebige Teilmenge seiner Deklarationen anderen Modulen zur Verfügung stellen, indem sie in der zugehörigen Schnittstelle veröffentlicht werden.

Übersetzungsreihenfolge

```
thales$ f
Greetings.od   Greetings.om   SayHello.od    SayHello.om
thales$ mmo -c makefile +t sayhello SayHello
thales$ make
oc   -c -u Greetings.od Greetings.om
oc   -c -u SayHello.od SayHello.om
oc    -o sayhello \
        Greetings.o SayHello.o
thales$
```

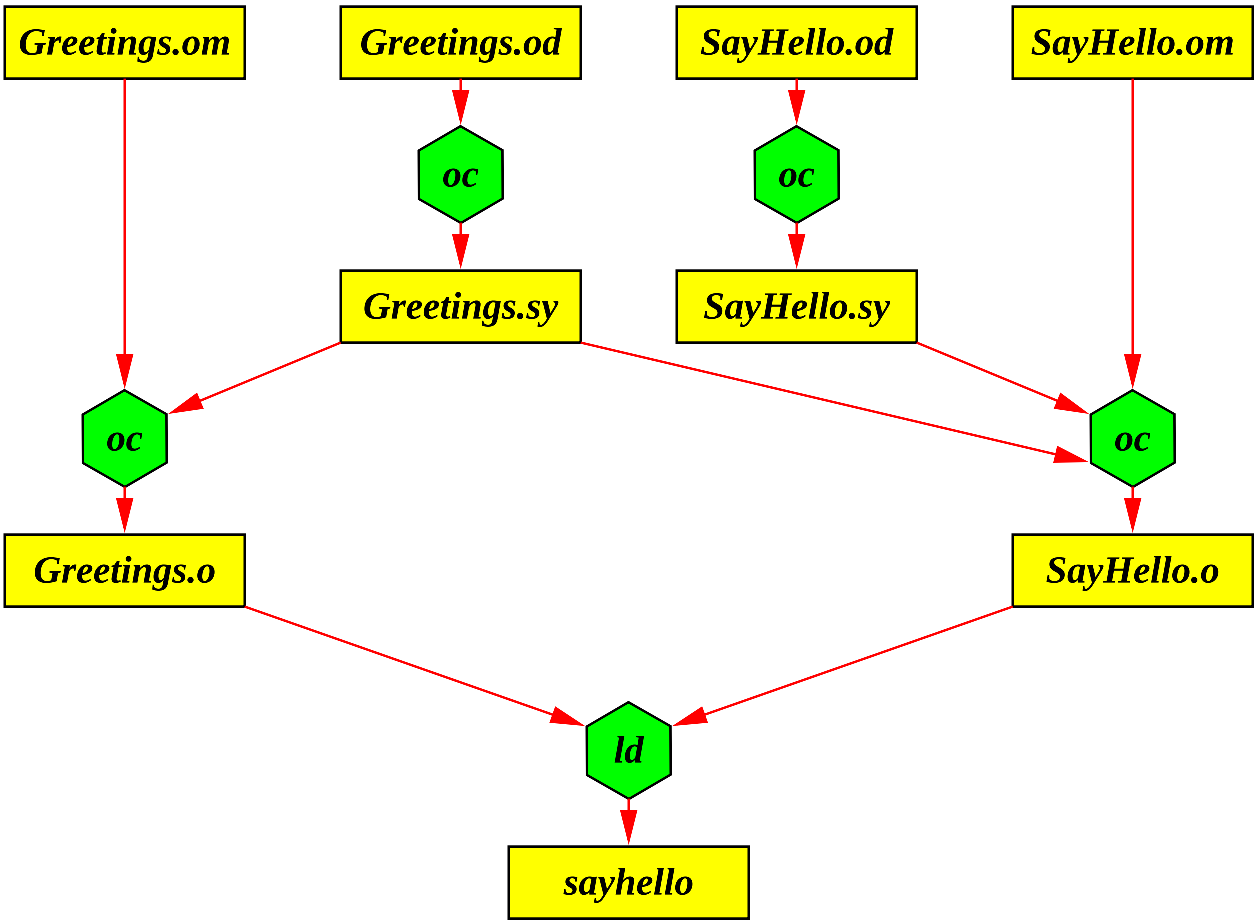
- Bevor eine Quelle (sei es eine Schnittstelle oder eine Implementierung) übersetzt werden kann, müssen alle Schnittstellen, die direkt oder indirekt verwendet werden, in übersetzter Form vorliegen.

- Wenn eine Schnittstelle (also eine Datei mit der Endung ".od") übersetzt wird, legt der Compiler das Resultat in einer gleichnamigen Datei mit der Endung ".sy" ab, die für "Symbol File" steht.

- Die erfolgreiche Übersetzung von Implementierungen führt zur Generierung von Objekten (Maschinen-Code, Datei-Endung ".o").

- Wenn alle Objekte vorliegen, können sie zu einem ausführbaren Programm zusammengebaut werden.

Übersetzungsabhängigkeiten



- Insgesamt gibt es in diesem Beispiel vier Übersetzungen (2 Schnittstellen und 2 Implementierungen) und einen Bindevorgang (wird von *ld*, dem *linkage editor* erledigt).

- Die Schnittstellen können in diesem Beispiel in beliebiger Reihenfolge übersetzt werden. Die Übersetzung der Implementierung von *Greetings* ist sofort möglich, wenn *Greetings.sy* vorliegt, während für die Übersetzung von *SayHello.om* sowohl *Greetings.sy* als auch *SayHello.sy* vorliegen muß.

Übersetzungsabhängigkeiten

- *mmo* ist ein Werkzeug, das durch die Inspektion der Quellen die Abhängigkeiten ermittelt und sie in eine für *make* verständliche Form im *makefile* ablegt.

- *make* ist ein Werkzeug, das dem *makefile* Abhängigkeiten zwischen Dateien und Prozeduren zur Herstellung bzw. Aktualisierung entnimmt und dann die jeweils notwendigen Prozeduren aufruft.

- So sehen die Abhängigkeiten innerhalb von *makefile* im vorgestellten Beispiel aus:

```
Greetings.sy:
Greetings.o:    Greetings.sy
SayHello.sy:
SayHello.o:     SayHello.sy Greetings.sy
```

Eine Abstraktion für Stapel

Stacks.od

```
DEFINITION Stacks;

   IMPORT Objects;

   TYPE
      Stack = POINTER TO StackRec;
      StackRec = RECORD (Objects.ObjectRec) END;
      Element = Objects.Object;

   PROCEDURE Create(VAR stack: Stack);
   PROCEDURE AddElement(stack: Stack; element: Element);
   PROCEDURE RemoveElement(stack: Stack);
   PROCEDURE GetCurrentElement(stack: Stack;
                               VAR element: Element);
   PROCEDURE Length(stack: Stack) : INTEGER;

END Stacks.
```

- So könnte eine Abstraktion für heterogene Stapel aussehen, die mit beliebigen Erweiterungen von *Objects.Object* gefüllt werden kann.

Der Basistyp *Objects.Object*

```
DEFINITION Objects;

   TYPE
      Object = POINTER TO ObjectRec;
      ObjectRec = RECORD END;

END Objects.
```

- In der Ulmer Oberon-Bibliothek dient der Typ *Objects.ObjectRec* als Basistyp für alle Record-Typen.
- Durch diese Konvention haben alle Record-Typen einen gemeinsamen Basistyp.
- Wenn diese Konvention konsequent eingehalten wird, dann können an Parameter vom Typ *Objects.Object* beliebige Zeiger auf Records übergeben werden.
- Aus diesem Grunde ist auch der Record-Typ *Stacks.StackRec* eine Erweiterung von *Objects.ObjectRec*.
- Dadurch können auch Stapel in einen Stapel eingefügt werden.

Information Hiding

Stacks.od

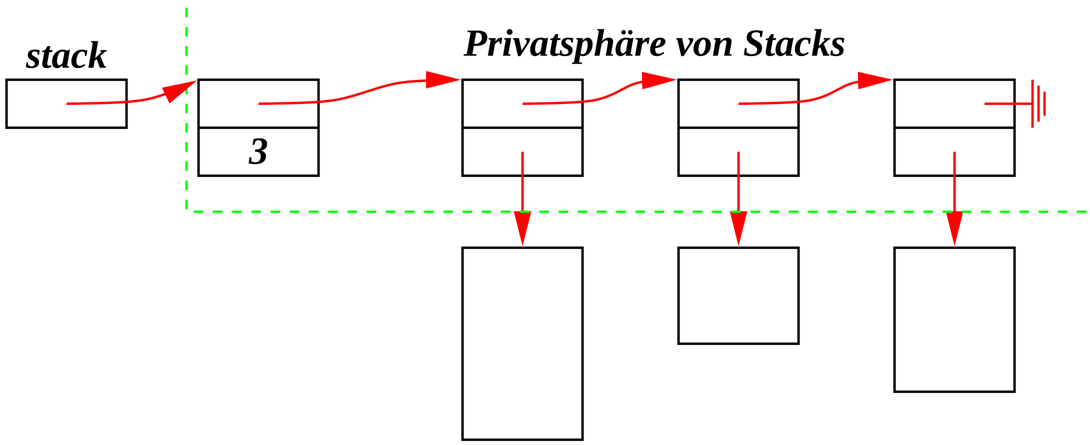
```
TYPE
   Stack = POINTER TO StackRec;
   StackRec = RECORD (Objects.ObjectRec) END;
   Element = Objects.Object;
```

Stacks.om

```
TYPE
   Stack = POINTER TO StackRec;
   LinkedNode = POINTER TO LinkedNodeRec;
   StackRec =
      RECORD
         (Objects.ObjectRec)
         top: LinkedNode;
         length: INTEGER;
      END;
   Element = Objects.Object;
   LinkedNodeRec =
      RECORD
         next: LinkedNode;
         element: Element;
      END;
```

- Grundsätzlich ist es üblich, Schnittstellen minimal zu halten.
- Dies ist auch bei der Veröffentlichung von Record-Typen möglich. In der Schnittstelle können – im Vergleich zur Implementierung – weniger (möglicherweise auch gar keine) Komponenten veröffentlicht werden.
- Dies verhindert effektiv den Zugriff externer Module auf die internen Datenstrukturen einer Implementierung.

Information Hiding



- Sichtbarkeitsgrenzen, die sich aus der Minimalisierung der Schnittstellen ergeben, können nahezu beliebig durch Datenstrukturen gezogen werden.

- Gegebenenfalls auch mitten durch Records hindurch, wenn Record-Komponenten partiell veröffentlicht worden sind oder mehrere Module (über Typ-Erweiterungen) private Komponenten zu einem ein- oder mehrfach erweiterten Record-Typ unterhalten.

- Durch die Minimalisierung der Schnittstellen wird sichergestellt, daß die internen Datenstrukturen modifiziert werden können, ohne daß dies andere Module betreffen kann.

Implementierung von *Stacks*

Stacks.om

```
PROCEDURE Create(VAR stack: Stack);
BEGIN
   NEW(stack); stack.top := NIL; stack.length := 0;
END Create;

PROCEDURE AddElement(stack: Stack; element: Element);
   VAR
      node: LinkedNode;
BEGIN
   NEW(node); node.element := element;
   node.next := stack.top; stack.top := node;
   INC(stack.length);
END AddElement;

PROCEDURE RemoveElement(stack: Stack);
BEGIN
   ASSERT(stack.length > 0);
   stack.top := stack.top.next;
   DEC(stack.length);
END RemoveElement;

PROCEDURE GetCurrentElement(stack: Stack;
                            VAR element: Element);
BEGIN
   ASSERT(stack.length > 0);
   element := stack.top.element;
END GetCurrentElement;

PROCEDURE Length(stack: Stack) : INTEGER;
BEGIN
   RETURN stack.length
END Length;
```

Literaturhinweise zu Modulen, Typ-Erweiterungen und Abstraktionen

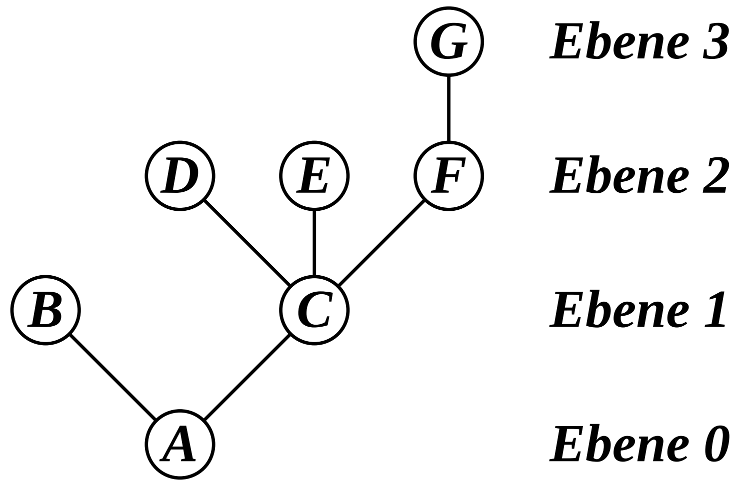
Im Rahmen dieser Vorlesung besteht leider nicht die Möglichkeit, das gelegte Fundament in Richtung fortgeschrittener OO-Techniken auszubauen. Folgende Verweise können dabei behilflich sein:

- Niklaus Wirth, “Type Extensions”, ACM Transactions on Programming Languages and Systems, Vol. 10, No. 2, April 1988, pp. 204-214.
- Bertrand Meyer, “Object-oriented Software Construction”, Prentice-Hall.
  Hier empfehle ich insbesondere auch dem Anfänger das erste Kapitel über “Aspects of software quality”, das außerordentlich lesenswert ist.
- Bertrand Meyer, “From Structured Programming to Object-Oriented Design: The Road to Eiffel”, Structured Programming, 1989, No. 1, pp. 19-39.
- Bertrand Meyer, “Reusability: The Cases for Object-Oriented Design”, IEEE Software, March 1987, Vol. 4, pp. 50-53.
- Andreas Borchert, “OO-Techniken in Oberon”, 2. Kapitel des Vorlesungsskripts zu “Entwicklung objektorientierter Bibliotheken”, WS 2000/2001
  `http://www.mathematik.uni-ulm.de/sai/ws00/oolib/`
  Oberon geht mit seinen Techniken seinen eigenen Weg. Wer diesen weiter erkunden möchte, dürfte in diesem Skript eine Einführung dazu und sehr viele Beispiele finden.

Bäume

- Bäume gehören zu den wichtigsten Datenstrukturen.
- Es gibt sie in sehr vielen Varianten, wovon wir nur einige im Rahmen dieser Vorlesung kennenlernen werden.
- Eine allgemeine Definition von Bäumen nach Knuth:
  Ein Baum ist eine endliche Menge *T* von einem oder mehreren Knoten, so daß
  - es einen speziell ausgezeichnenten Knoten gibt, der als *Wurzel* bezeichnet wird: **root**(*T*); und
  - die verbleibenden Knoten (ohne die Wurzel) in $m \geq 0$ disjunkte Mengen $T_1, ..., T_m$ aufgeteilt sind, wobei jede dieser Mengen wiederum ein Baum ist. Die Bäume $T_1, ..., T_m$ werden die *Unterbäume* der Wurzel genannt.
- Diese Definition ist rekursiv. Allerdings ist es kein Zirkelschluß, da die Gesamtmenge der Knoten endlich ist und sich die Menge der Knoten bei jedem Rekursionsschritt um mindestens eins vermindert, so daß am Ende Bäume mit nur einem Knoten verbleiben.
- Bäume lassen sich auch nicht-rekursiv definieren, jedoch paßt Rekursion zur Natur der Bäume, da viele Algorithmen auf Bäumen rekursiv arbeiten.
- Hinweis: Diese Seite und die folgenden lehnen sich sehr eng an das Kapitel 2.3 von “The Art of Computer Programming”, Band 1, von Donald E. Knuth an. Insbesondere wurden auch einige Abbildungen mehr oder weniger direkt übernommen.

Bäume



- Jeder Knoten eines Baumes ist die Wurzel eines der Unterbäume innerhalb des Baumes.

- Die Zahl der Unterbäume eines Knotens wird als *Grad* bezeichnet.

- Ein Knoten mit dem Grad 0 wird als *Endknoten* oder als Blatt bezeichnet.

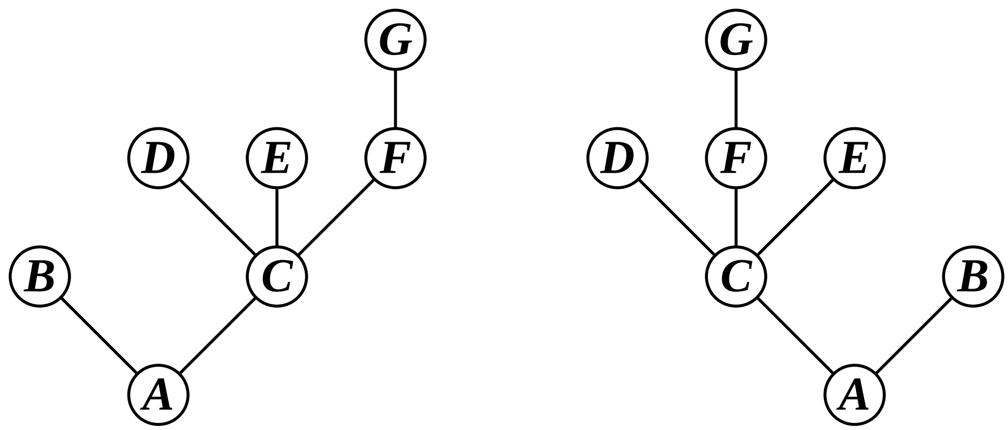
- *Verzweigungsknoten* sind Knoten mit einem Grad > 0.

- Die *Ebene* eines Knotens in Bezug auf T wird rekursiv definiert:

  - Die Ebene von **root**(T) ist 0.

  - Die Ebene eines anderen Knotens ist um 1 größer als die Ebene des Knotens, der die Wurzel des unmittelbar übergeordneten Unterbaumes ist.

Geordnete und orientierte Bäume



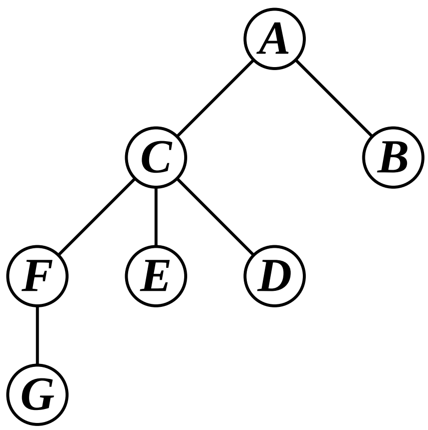
- Wenn die relative Ordnung der Unterbäume $T_1, ..., T_m$ relevant ist, wird von einem *geordneten Baum* gesprochen. Entsprechend ist dann beispielsweise T_2 der zweite Unterbaum der Wurzel.

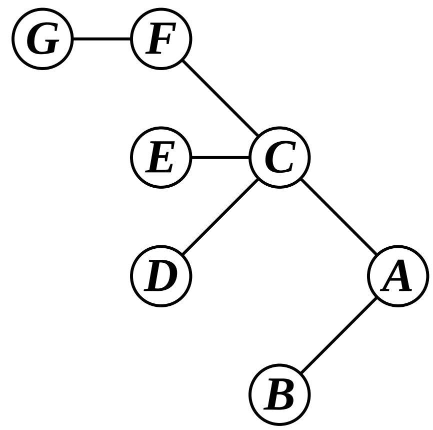
- Wenn hingegen die Ordnung keine Rolle spielt, wird von *orientierten Bäumen* gesprochen.

- Die beiden Bäume im obigen Beispiel sind verschieden, wenn sie als geordnete Bäume betrachtet werden, jedoch äquivalente Darstellungen, wenn sie als orientierte Bäume zu verstehen sind.

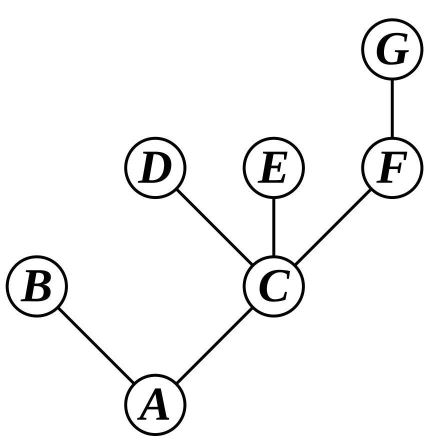
- Normalerweise sind alle Bäume geordnet, da sich das relativ naturgemäß aus der Datenstruktur im Rechner ergibt.

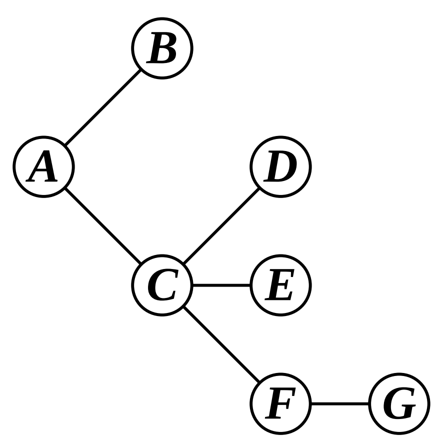
Darstellung von Bäumen



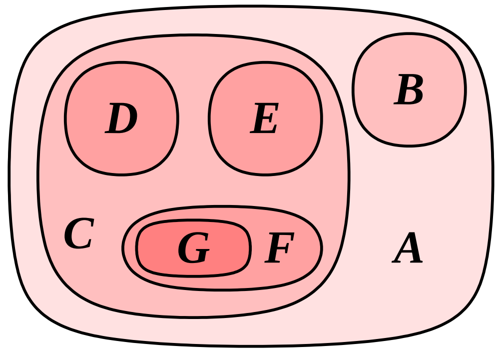


- Bäume können in vielerlei Weisen dargestellt werden.
- Da häufig von “über”, “höher als” bei Knoten gesprochen wird, empfiehlt es sich, von einer einheitlichen Darstellungsweise auszugehen.
- Obwohl Bäume in der Natur von unten nach oben wachsen, hat sich diese Vorstellung bei den Informatikern nicht durchgesetzt. Stattdessen ist üblicherweise die Wurzel ganz oben (siehe Diagramm oben links) und von dort wächst der Baum nach unten.





Darstellung von Bäumen

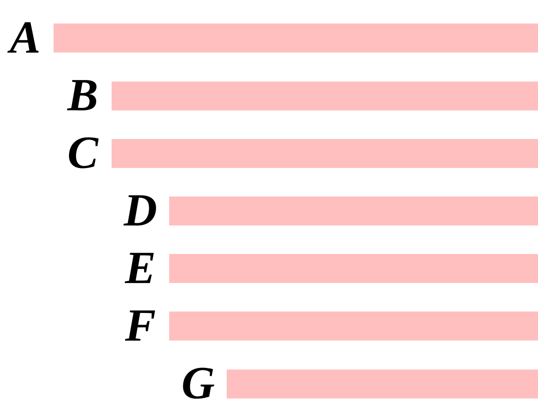


- Es gibt zahlreiche weitere Möglichkeiten, Bäume darzustellen:

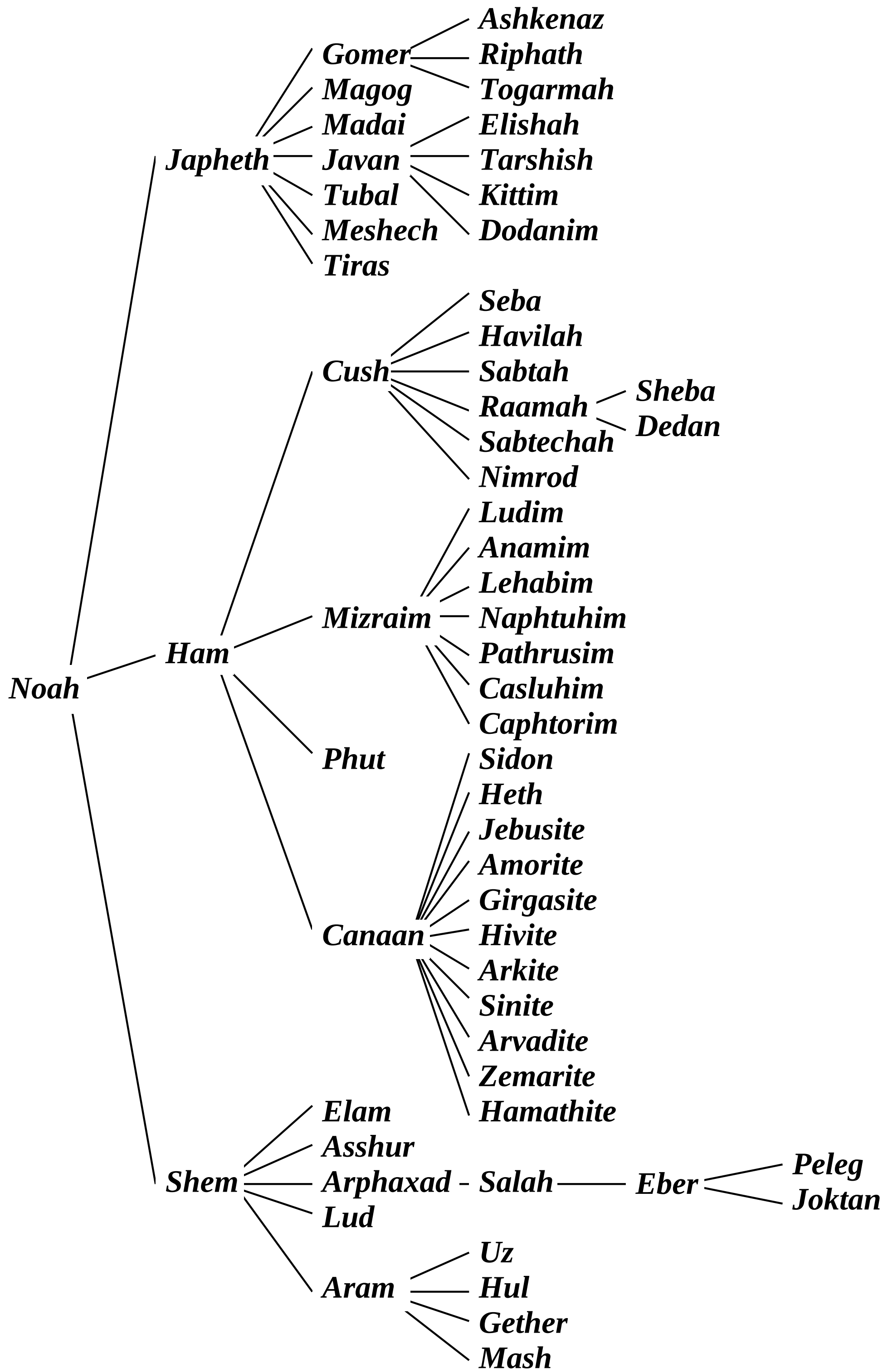
  - Die Darstellung in Form verschachtelter Mengen ist insbesondere für orientierte Bäume geeignet.

  - Für die Eingabe ist eine Syntax denkbar, die mit Klammern die Verschachtelung vorgibt:
    `(A(B)(C(D)(E)(F(G))))`

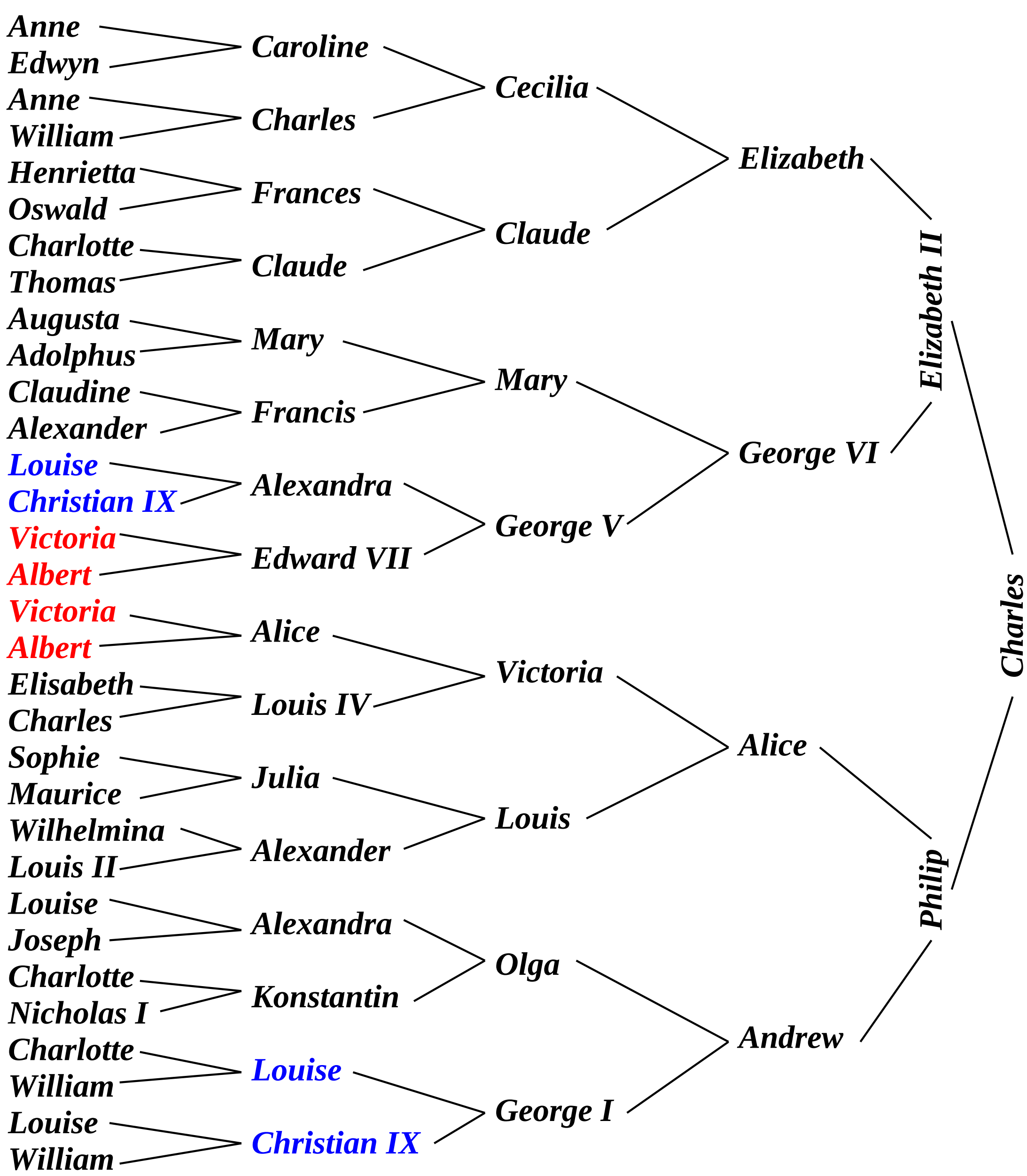
  - Bäume entsprechen Hierarchien und entsprechend können auch geordnete Bäume so dargestellt werden, daß für jeden Knoten eine entsprechend eingerückte Zeile verwendet wird (je höher die Ebene, desto weiter ist es einzurücken).



Familienbäume: Nachfahren



Familienbäume: Vorfahren



Familienbäume

- Viele Bezeichnungen bei Bäumen lehnen sich an die der Familienbäume an.

- Es gibt zwei Varianten von Familienbäumen:

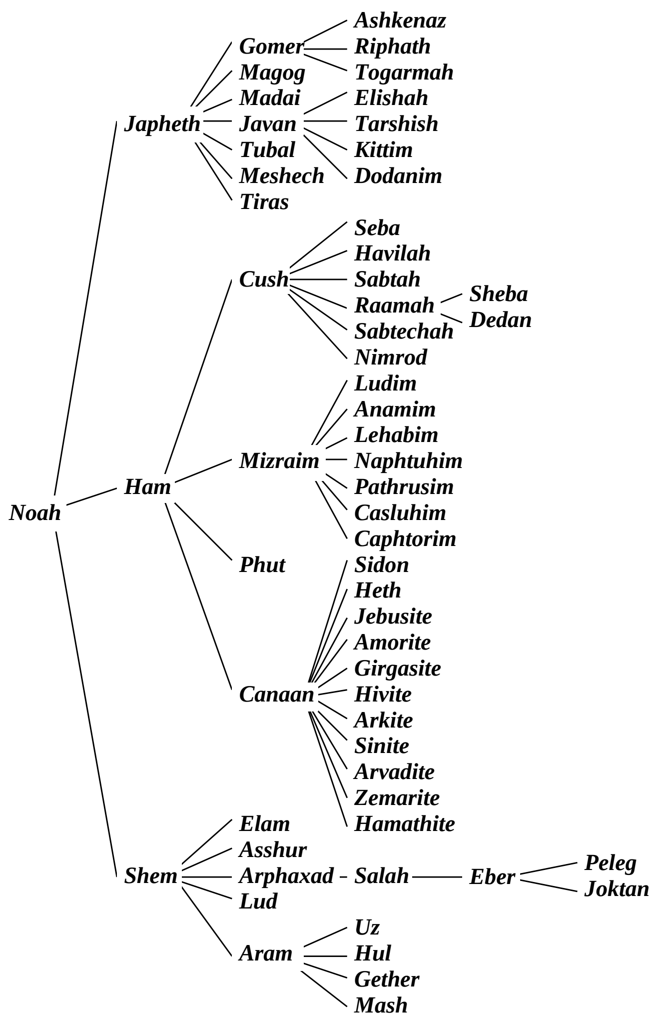
  - Ausgehend von einer Wurzel werden alle Nachfahren (bis zu einer gewissen Ebene) angegeben.

  - Ausgehend von einer Wurzel werden alle Vorfahren (bis zu einer gewissen Ebene) aufgeführt.

- Familienbäume sind nicht notwendigerweise Bäume im Sinne der Definition, wenn es zu Eheschließungen innerhalb der Verwandtschaft kommt. So zählt sowohl Elizabeth II als auch Philip das Ehepaar von Königin Victora und Prinz Albert zu ihren Ururgroßeltern (siehe rote Farbe). Und auch König Christian IX und Königin Louise zählen beidseitig zu den Vorfahren (siehe blaue Farbe).

- Das Problem läßt sich beheben, wenn die Knoten nicht die jeweiligen Personen repräsentieren, sondern die jeweilige Rolle (z.B. als Großvater mütterlicherseits).

Von Familienbäumen abgeleitete Terminologie



- Die Wurzel wird als *Elternteil* der Wurzeln ihrer Teilbäume bezeichnet.

- Letztere sind *Kinder* ihres Elternteils bzw. *Geschwister* untereinander.

- Die Begriffe *Vorfahre* und *Nachfahre* bezeichnen Beziehungen, die sich über mehrere Ebenen erstrecken können.

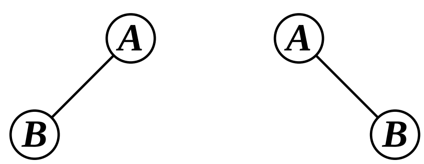
Binäre Bäume

- Ein binärer Baum ist eine endliche Menge von Knoten, die entweder leer ist oder aus einer Wurzel und den Elementen zweier disjunkter binärer Bäume besteht, die linker und rechter Teilbaum der Wurzel genannt werden.

- Binäre Bäume sind **kein** Spezialfall eines generellen Baumes:

  - So ist die leere Menge als binärer Baum zulässig, jedoch nicht als allgemeiner Baum, der immer eine Wurzel besitzt.

  - Folgende zwei binäre Bäume sind nicht identisch:



    In dem einen Fall liegt nur ein linker Teilbaum vor, in dem anderen Fall nur ein rechter.

- Der Familienbaum mit den Vorfahren (*Pedigree*) ist ein binärer Baum. Ein weiteres Beispiel findet sich bei den Ausscheidungskämpfen bei Tennisturnieren: Der Gewinner ist die Wurzel, auf der 1. Ebene sind die Teilnehmer des Finales, auf der 2. Ebene die der Halbfinale usw.

Repräsentierung binärer Bäume

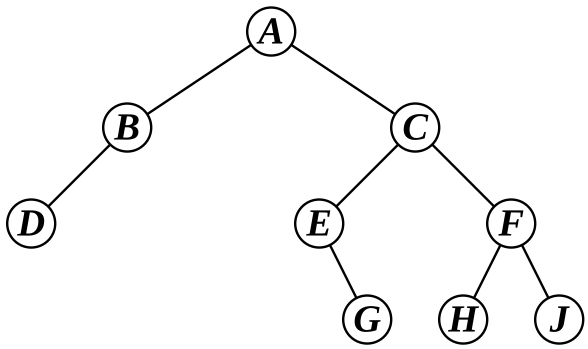
- Binäre Bäume lassen sich sehr bequem als Datentyp realisieren:

  - Es gibt einen Zeiger auf die Wurzel namens *root*. Wenn der Zeiger **NIL** ist, dann ist der binäre Baum leer.

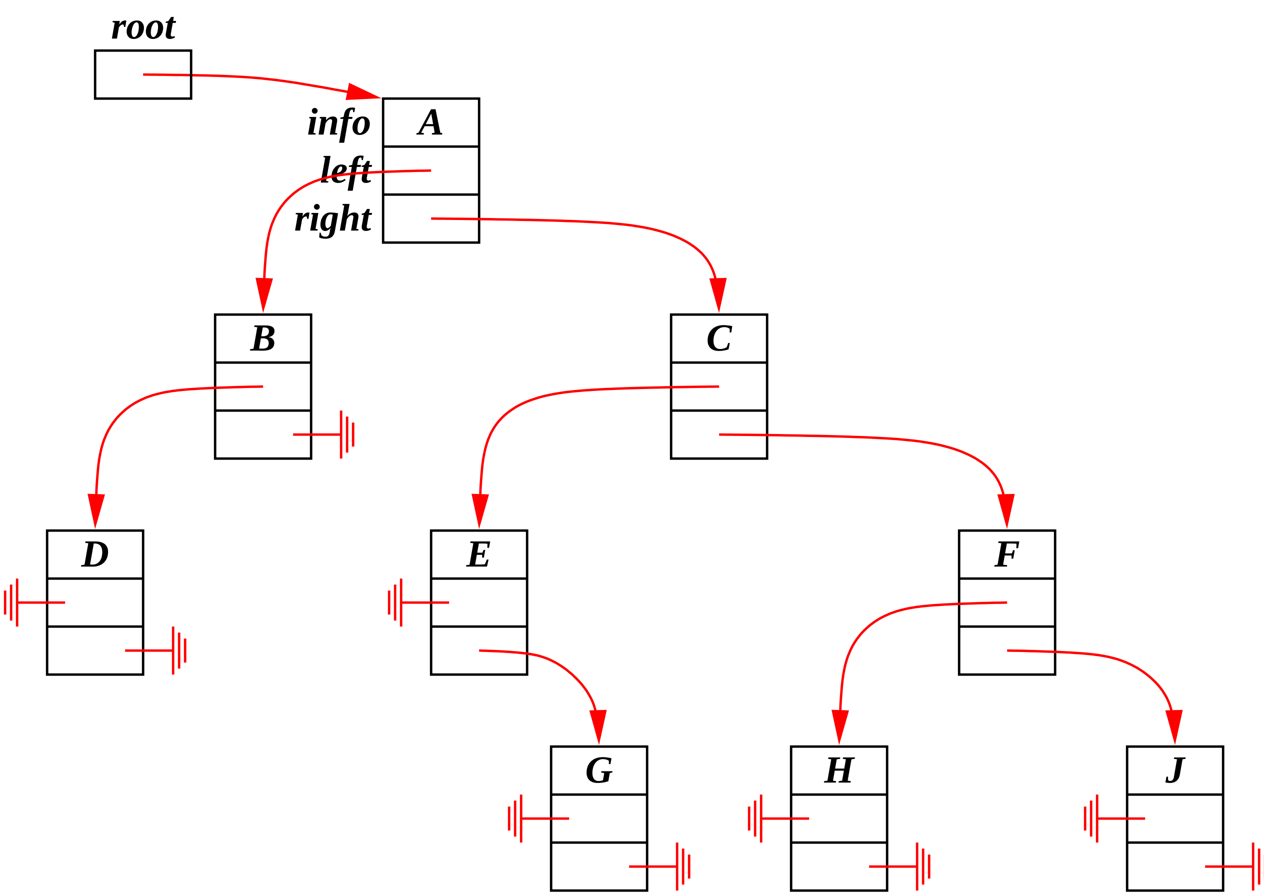
  - Ein Knoten des binären Baumes besteht aus der ihm zugeordneten Information (entweder ein Zeiger namens *object* oder ein Record namens *info*) und den beiden Zeigern auf den linken und rechten Teilbaum.

```
TYPE
   Node = POINTER TO NodeRec;
   NodeRec =
      RECORD
         object: Objects.Object;
         left, right: Node;
      END;
   BinaryTree = POINTER TO BinaryTreeRec;
   BinaryTreeRec =
      RECORD
         (Objects.ObjectRec)
         root: Node;
      END;
```

Repräsentierung binärer Bäume



- So sieht die Repräsentierung obigen binären Baumes aus:



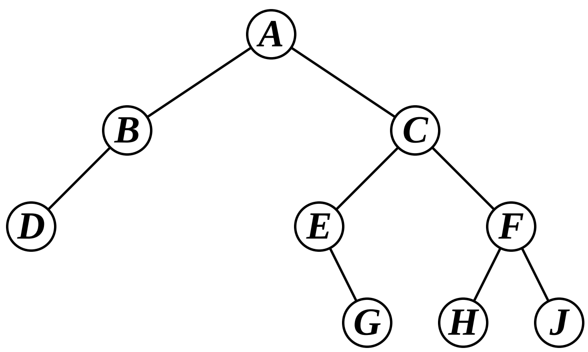
Traversierung binärer Bäume

- Es gibt drei prinzipielle Möglichkeiten, einen binären Baum zu traversieren: *preorder*, *inorder* und *postorder*.
- Wenn der binäre Baum leer ist (d.h. gleich **NIL** ist), gibt es bezüglich der Traverse nichts weiter zu tun.
- Ist der Baum nicht leer, so definieren sich die einzelnen Traversen wie folgt:

preorder	Besuche die Wurzel Traversiere den linken Teilbaum Traversiere den rechten Teilbaum
inorder	Traversiere den linken Teilbaum Besuche die Wurzel Traversiere den rechten Teilbaum
postorder	Traversiere den linken Teilbaum Traversiere den rechten Teilbaum Besuche die Wurzel

```
PROCEDURE TraverseInorder(node: Node);
BEGIN
   IF node # NIL THEN
      TraverseInorder(node.left);
      Visit(node);
      TraverseInorder(node.right);
   END;
END TraverseInorder;
```

Traversierung binärer Bäume



- Folgende Reihenfolgen ergeben sich, wenn die Knoten entsprechend den vorgestellten Traversen besucht werden:

preorder	A B D C E G F H J
inorder	D B A E G C H F J
postorder	D B G E H J F C A

Sortierte binäre Bäume

- Ein Tupel (T, v, V, R) ist ein *sortierter binärer Baum*

  - bezüglich der Wertefunktion $v : T \to V$ und

  - bezüglich der vollständigen Ordnungsrelation R für $V \times V$,

  wenn T ein binärer Baum ist, der entweder leer ist oder für den bezüglich des linken Teilbaums T_1 und des rechten Teilbaums T_2 gilt, daß

  - $v(\text{root}(T)) >_R v(t) \quad \forall\, t \in T_1$,

  - $v(\text{root}(T)) \leq_R v(t) \quad \forall\, t \in T_2$ und daß

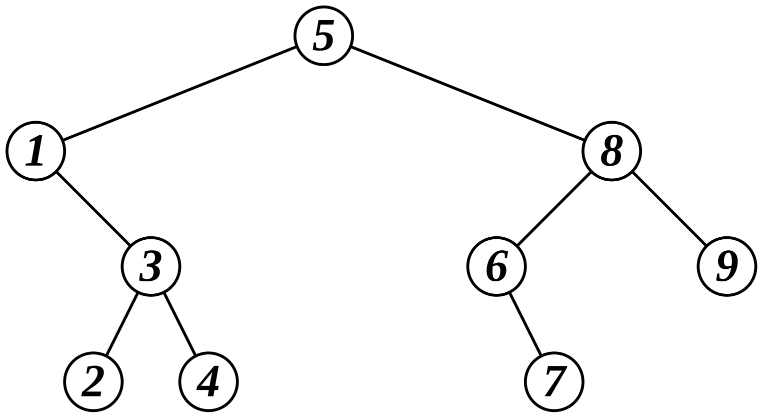
  - sowohl (T_1, v, V, R) als auch (T_2, v, V, R) sortierte binäre Bäume sind.

- Gelegentlich ist es auch sinnvoll, darauf zu bestehen, daß

$$v(t_1) \neq_R v(t_2)\, \forall\, t_1, t_2 \in T, t_1 \neq t_2$$

  In diesem Falle ist die Wertefunktion $v : T \to V_T$ bijektiv, wobei $V_T = \{v \mid \exists t \in T : V(t) = v\}$, d.h. mit Hilfe eines Wertes $v \in V_T$ kann ein Knoten $t \in T$ eindeutig bestimmt werden.

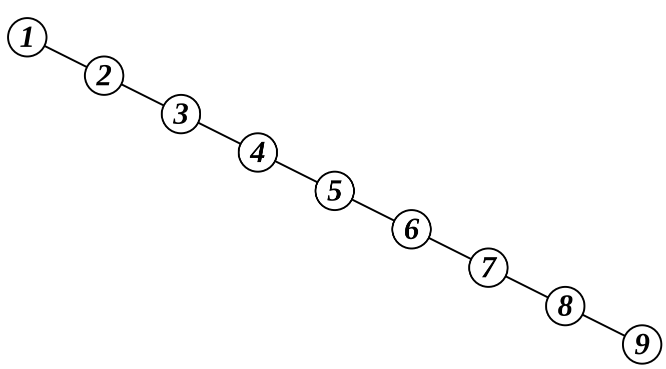
Sortierte binäre Bäume



- Wenn ein sortierter binärer Baum in *inorder* traversiert wird, dann werden alle Werte in sortierter Form durchlaufen.

- Die Höhe $h : T \to \mathbb{N}_0$ eines Baumes definiert sich wie folgt: Falls T leer ist, dann ist $h(T) = 0$, ansonsten gilt $h(T) = max(h(T_1), h(T_2)) + 1$.

- $\lceil \log_2(n+1) \rceil \leq h(T) \leq n$ mit $n = \text{card}(T)$ (Anzahl der Elemente in T).



Ausgeglichenheit

- Ein Baum T ist *in der Höhe ausgeglichen*, wenn entweder T leer ist oder gilt, daß

  - $\mid h(T_1) - h(T_2) \mid \leq 1$ und

  - sowohl T_1 als auch T_2 in der Höhe ausgeglichen sind.

- Ein Baum T ist *nach dem Gewicht ausgeglichen*, wenn entweder T leer ist oder gilt, daß

  - $\mid \text{card}(T_1) - \text{card}(T_2) \mid \leq 1$ und

  - sowohl T_1 als auch T_2 nach dem Gewicht ausgeglichen sind.

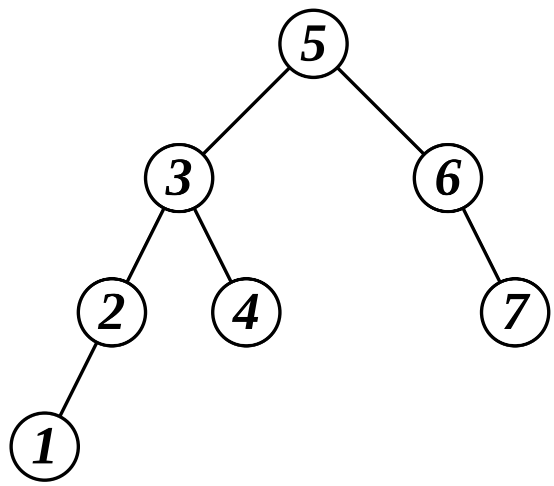
- Wenn ein Baum T nach dem Gewicht ausgeglichen ist, so folgt daraus, daß er auch in der Höhe ausgeglichen ist.

- Nach dem Satz von Adelson-Velsky und Landis gilt für einen in der Höhe ausgeglichenen Baum T:
  $\lceil \log_2(n+1) \rceil \leq h(T) \leq 1.4404 \log_2(n+2) - 0.3277$

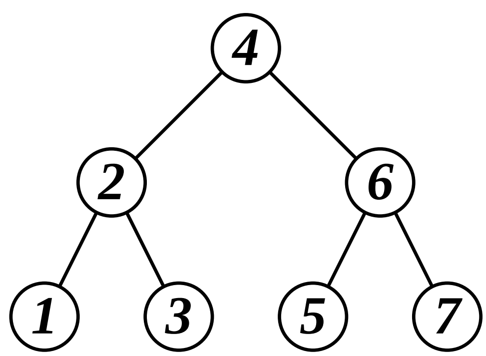
- Somit sind Suchpfade im höhenausgeglichene Bäume niemals um mehr als 45% länger im Vergleich zum optimalen Fall.

Ausgeglichenheit

- In der Höhe ausgeglichen, jedoch nicht nach dem Gewicht:



- Nach dem Gewicht ausgeglichen (und auch in der Höhe):



Suche in einem binären sortierten Baum

- Gegeben sei ein Wert $k \in V$ und gesucht wird ein Knoten $t \in T$ mit $v(t) = k$:

  1. Wenn T leer ist, dann ist der gesuchte Wert nicht vorhanden.

  2. Sei $t = \text{root}(T)$. Wenn $v(t) = k$, dann ist der gesuchte Wert gefunden.

  3. Es geht weiter mit dem ersten Schritt, wobei

     - $T = T_1$, falls $k < v(t)$, und

     - $T = T_2$, falls $k \geq v(t)$.

- Diese Suche hat einen Aufwand von $O(h(T))$.

Einfügen in einen binären sortierten Baum

- Gegeben sei ein neues Element u mit $k = v(u)$, das in T einzufügen ist:

  1. Erzeuge einen neuen binären sortieren Baum U, der nur aus der Wurzel u besteht.

  2. Wenn T leer ist, dann ist T durch U zu ersetzen.

  3. Es geht weiter mit dem zweiten Schritt, wobei

     - $T = T_1$, falls $k < v(t)$, und

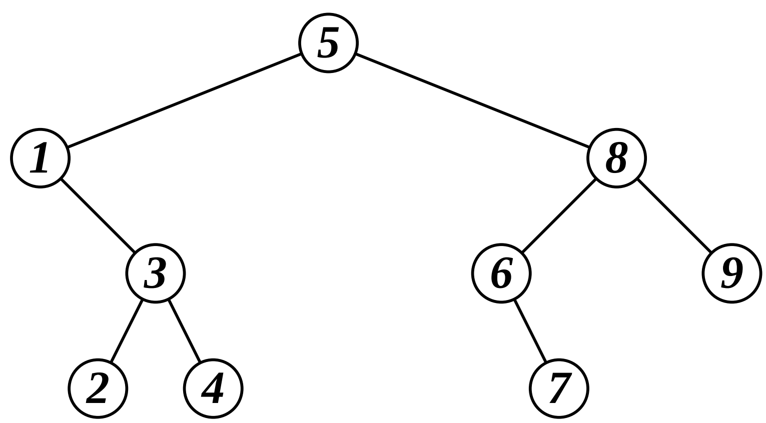
     - $T = T_2$, falls $k \geq v(t)$.

- Das Einfügen hat einen Aufwand von $O(h(T))$.

- Wenn Elemente in sortierter Reihenfolge eingefügt werden, degeneriert der Baum zur linearen Liste mit $h(T) = \text{card}(T)$. Dieses "*worst case*"-Szenario läßt sich durch diverse Algorithmen vermeiden, die den Baum bei Bedarf umbauen (z.B. die Höhenausgeglichenheit sicherstellen).

Einfügen in einen binären sortierten Baum

- Wenn in einen leeren Baum T nacheinander 5, 1, 3, 2, 8, 4, 6, 7 und 9 eingefügt werden, sieht es so aus:



- Das erste Element, das eingefügt wird, bleibt die Wurzel.

- Jedes neue Element, das eingefügt wird, bildet zunächst einen Endknoten.

Löschen in einem binären sortierten Baum

- Der Knoten $u \in T$ ist aus T zu entfernen:

  1. Sei U der Teilbaum von T mit $\text{root}(U) = u$.

  2. Falls U_1 leer ist, dann ist U durch U_2 zu ersetzen.

  3. Falls U_2 leer ist, dann ist U durch U_1 zu ersetzen.

  4. Wenn jedoch U_1 und U_2 nicht leer sind, dann

     - ist $u_2 \in U_2$ so auszuwählen, daß $v(u_2) \leq v(t)\ \forall t \in U_2$ (einfach nur links absteigen, bis es nicht weitergeht),

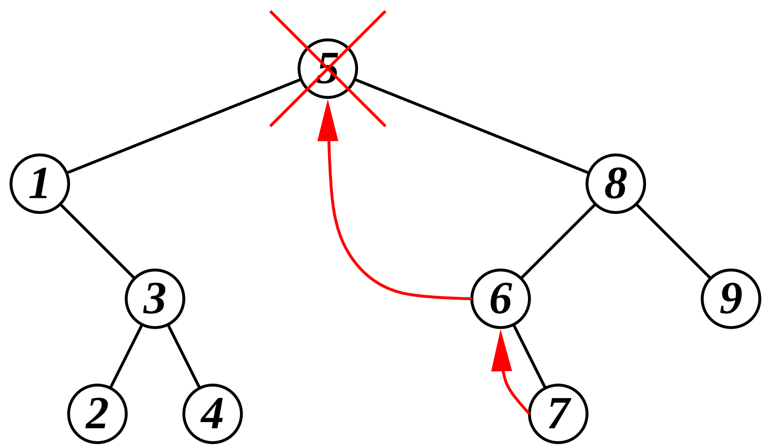
     - u_2 aus U_2 zu entfernen (recht einfach, da zumindest der linke Teilbaum leer ist) und

     - u_2 als Ersatz für den Knoten u zu verwenden.

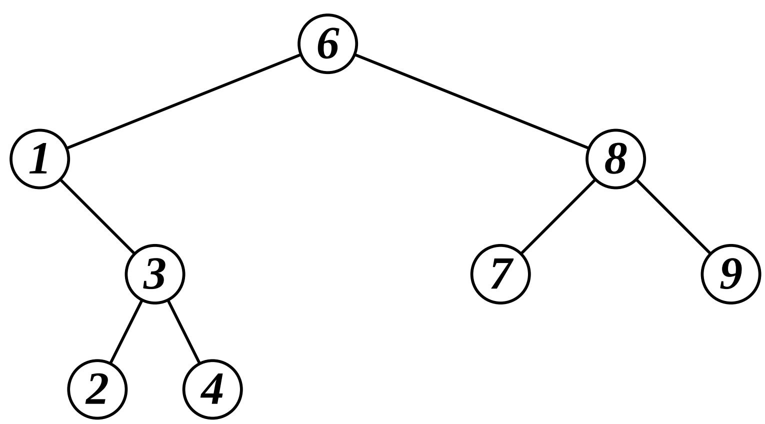
- Das Löschen hat einen Aufwand von $O(h(T))$, der im wesentlichen durch die Suche nach dem minimalen Element in U_2 bestimmt ist.

Löschen in einem binären sortierten Baum

- Wenn die 5 zu löschen ist, dann ist U_2 der rechte Teilbaum mit $\{6, 7, 8, 9\}$, wovon 6 das kleinste Element ist:



- Entsprechend wird die 6 aus dem rechten Teilbaum entfernt und als Ersatz für die 5 verwendet:



Nächstes Element in einem binären sortierten Baum

- Gegeben sei ein Wert $k \in V$, gesucht wird der Knoten $n(T, k) = u \in T$ mit $v(u) > k$ und $\nexists u' \in T : v(u') > k \wedge v(u') < v(u)$.

  1. Wenn T leer ist, dann gibt es kein solches Element.

  2. Sei $t = \text{root}(T)$.

  3. Falls $k \geq v(t)$, dann ist $n(T_2, k)$ das gewünschte Element (falls existent).

  4. Falls hingegen $k < v(t)$, dann ist es $n(T_1, k)$, falls existent, oder andernfalls t.

- Das erste Element eines binären sortierten Baumes läßt sich leicht bestimmen, indem ganz nach links abgestiegen wird.

Prozedurtypen

SortedBinaryTrees.od

```
TYPE
   CompareProc = PROCEDURE
         (object1, object2: Objects.Object) : INTEGER;
      (* return an integer value
            < 0 if object1 < object2
            = 0 if object1 = object2
            > 0 if object1 > object2
      *)
```

- Wenn ein abstraktes Modul für sortierte binäre Bäume formuliert wird, sollte es nach Möglichkeit von dem zu vergleichenden Typ V und der vollständigen Ordnungsrelation R unabhängig sein.

- Das läßt sich in Oberon durch die Verwendung von Prozedurtypen erreichen. Einer Variablen von einem Prozedurtyp (hier z.B. *CompareProc*) kann eine beliebige Prozedur zugewiesen werden, die einen äquivalenten Prozedurkopf besitzt (hier z.B. *Compare*).

CalendarManager.om

```
PROCEDURE Compare(app1, app2: Objects.Object) : INTEGER;
BEGIN
   RETURN Op.Compare(app1(Appointment).time,
                     app2(Appointment).time)
END Compare;
```

Homogenität eines sortierten binären Baumes

SortedBinaryTrees.od

```
TYPE
   ComparableProc =
         PROCEDURE (object: Objects.Object) : BOOLEAN;

PROCEDURE Create(VAR tree: Tree;
                 compare: CompareProc;
                 comparable: ComparableProc);
   (* creates a binary tree that
      - requires all inserted objects to be comparable, and
      - sorts its nodes according to compare
   *)
```

- Im Vergleich zu unsortierten Datenstrukturen muß bei sortierten Kollektionen darauf geachtet werden, daß die vorgegebene vollständige Ordnungsrelation für alle einzufügenden Objekte anwendbar ist.

- Bei dem Konstruktor *SortedBinaryTrees.Create* wird dann neben der Vergleichsoperation *compare* auch die Überprüfungsprozedur *comparable* mit übergeben, die sicherstellt, daß nur Objekte aufgenommen werden, die auch vergleichbar sind.

CalendarManager.om

```
PROCEDURE Comparable(object: Objects.Object) : BOOLEAN;
BEGIN
   RETURN object IS Appointment
END Comparable;
```

Schnittstelle für sortierte binäre Bäume

SortedBinaryTrees.od

```
TYPE
   Tree = POINTER TO TreeRec;
   TreeRec = RECORD (Objects.ObjectRec) END;
TYPE
   VisitProc = PROCEDURE (object: Objects.Object);
      (* called by Traverse for all objects of a tree *)

PROCEDURE Acceptable(tree: Tree;
             object: Objects.Object) : BOOLEAN;
   (* returns TRUE if object may be inserted into tree,
      i.e. if it is comparable to other objects of that
      tree and if there is no object yet in tree that
      is identical to object
   *)

PROCEDURE Comparable(tree: Tree;
                     key: Objects.Object) : BOOLEAN;
   (* returns TRUE if key can be compared
      to objects of tree
   *)

PROCEDURE Insert(tree: Tree; object: Objects.Object);
   (* precondition: object must be acceptable for tree;
      inserts object into tree
   *)

PROCEDURE Delete(tree: Tree; key: Objects.Object);
   (* precondition: key must be comparable to
      other objects of tree;
      deletes object out of the tree that is identical
      to key if present
   *)
```

Schnittstelle für sortierte binäre Bäume

SortedBinaryTrees.od

```
PROCEDURE Lookup(tree: Tree; key: Objects.Object;
                 VAR object: Objects.Object) : BOOLEAN;
   (* precondition: key must be comparable for tree;
      stores into object that object of the tree
         that is identical to key if present;
      returns TRUE if one object has been found
   *)

PROCEDURE First(tree: Tree;
                VAR object: Objects.Object) : BOOLEAN;
   (* returns first object in tree *)

PROCEDURE Last(tree: Tree;
               VAR object: Objects.Object) : BOOLEAN;
   (* returns last object in tree *)

PROCEDURE Next(tree: Tree; key: Objects.Object;
               VAR object: Objects.Object) : BOOLEAN;
   (* returns lowest object in tree > key *)

PROCEDURE Prev(tree: Tree; key: Objects.Object;
               VAR object: Objects.Object) : BOOLEAN;
   (* returns highest object in tree < key *)

PROCEDURE Traverse(tree: Tree; visit: VisitProc);
   (* call visit for all objects in tree in sorted order *)
```

Datenstruktur der Implementierung

SortedBinaryTrees.om

```
TYPE
   Node = POINTER TO NodeRec;
   NodeRec =
      RECORD
         object: Objects.Object;
         left, right: Node;
      END;
   Tree = POINTER TO TreeRec;
   TreeRec =
      RECORD
         (Objects.ObjectRec)
         compare: CompareProc;
         comparable: ComparableProc;
         root: Node;
      END;
```

- Die Knoten sind gegenüber den unsortierten binären Bäumen unverändert.
- Die bei *Create* übergebenen Prozedurvariablen werden im Record für den Baum zur späteren Verwendung notiert.

Binäre Suche im Baum

SortedBinaryTrees.om

```
PROCEDURE Find(tree: Tree; key: Objects.Object;
               VAR parent, node: Node) : BOOLEAN;
BEGIN
   node := tree.root; parent := NIL;
   WHILE node # NIL DO
      IF tree.compare(key, node.object) = 0 THEN
         RETURN TRUE
      END;
      parent := node;
      IF tree.compare(key, node.object) < 0 THEN
         node := node.left;
      ELSE
         node := node.right;
      END;
   END;
   RETURN FALSE
END Find;
```

- Ein Abstieg entsprechend einem Schlüssel ist an mehreren Stellen notwendig (*Insert*, *Delete* und *Lookup*), so daß es sich lohnt, dies innerhalb von *SortedBinaryTrees* nur einmal zu formulieren.

- Der jeweilige Elternteil wird mit zurückgegeben, um ggf. das Einfügen zu erleichtern.

Akzeptanz und Vergleichbarkeit

SortedBinaryTrees.om

```
PROCEDURE Acceptable(tree: Tree;
                     object: Objects.Object) : BOOLEAN;
   (* returns TRUE if object may be inserted into tree,
      i.e. if it is comparable to other objects of that tree
      and if there is no object yet in tree that is
      identical to object
   *)
   VAR
      parent, node: Node; (* not used *)
BEGIN
   RETURN tree.comparable(object) &
          ~Find(tree, object, parent, node)
END Acceptable;

PROCEDURE Comparable(tree: Tree;
                     key: Objects.Object) : BOOLEAN;
   (* returns TRUE if key can be compared to
      objects of tree
   *)
BEGIN
   RETURN tree.comparable(key)
END Comparable;
```

- Dieses Modul besteht darauf, daß jeder Vergleichswert innerhalb eines Baumes eindeutig sein muß.
- Das hat den Vorteil, daß *Delete* und *Lookup* eindeutig definiert sind.

Suchen im sortieren binären Baum

SortedBinaryTrees.om

```
PROCEDURE Lookup(tree: Tree; key: Objects.Object;
                 VAR object: Objects.Object) : BOOLEAN;
   (* precondition: key must be comparable for tree;
      stores into object that object of the tree
         that is identical to key if present;
      returns TRUE if one object has been found
   *)
   VAR
      parent, node: Node;
BEGIN
   ASSERT(tree.comparable(key));
   IF Find(tree, key, parent, node) THEN
      object := node.object;
      RETURN TRUE
   ELSE
      RETURN FALSE
   END;
END Lookup;
```

Einfügen in den sortierten binären Baum

SortedBinaryTrees.om

```
PROCEDURE Insert(tree: Tree; object: Objects.Object);
   (* precondition: object must be acceptable for tree;
      inserts object into tree
   *)
   VAR
      found: BOOLEAN;
      parent, node: Node;
BEGIN
   ASSERT(tree.comparable(object));
   found := Find(tree, object, parent, node); ASSERT(~found);
   NEW(node); node.object := object;
   node.left := NIL; node.right := NIL;
   IF parent = NIL THEN
      tree.root := node;
   ELSIF tree.compare(object, parent.object) < 0 THEN
      parent.left := node;
   ELSE
      parent.right := node;
   END;
END Insert;
```

Löschen in einem sortierten binären Baum

SortedBinaryTrees.om

```
PROCEDURE Delete(tree: Tree; key: Objects.Object);
   (* precondition: key must be comparable to
      other objects of tree;
      deletes object out of the tree that is identical
      to key if present
   *)
   VAR
      parent, node: Node;

   PROCEDURE DeleteNode(VAR node: Node);
      (* ... naechste Folie ... *)
   END DeleteNode;

BEGIN (* Delete *)
   ASSERT(tree.comparable(key));
   IF ~Find(tree, key, parent, node) THEN RETURN END;
   IF parent = NIL THEN
      DeleteNode(tree.root);
   ELSIF parent.left = node THEN
      DeleteNode(parent.left);
   ELSE
      DeleteNode(parent.right);
   END;
END Delete;
```

- Zu beachten ist hier, daß der *node*-Parameter von *DeleteNode* ein **VAR**-Parameter ist. Entsprechend kann der verweisende Zeiger des Elternteils modifiziert werden.

Löschen in einem sortierten binären Baum

SortedBinaryTrees.om

```
PROCEDURE DeleteNode(VAR node: Node);
   VAR
      parent, lowest: Node;
BEGIN
   IF node.left = NIL THEN
      node := node.right;
   ELSIF node.right = NIL THEN
      node := node.left;
   ELSE
      (* find lowest node on right branch ... *)
      lowest := node.right; parent := node;
      WHILE lowest.left # NIL DO
         parent := lowest; lowest := lowest.left;
      END;
      (* ... and move that upwards *)
      node.object := lowest.object;
      IF lowest = node.right THEN
         DeleteNode(node.right);
      ELSE
         DeleteNode(parent.left);
      END;
   END;
END DeleteNode;
```

Iterationen im sortierten binären Baum

SortedBinaryTrees.om

```
PROCEDURE First(tree: Tree;
                VAR object: Objects.Object) : BOOLEAN;
   (* returns first object in tree *)
   VAR
      node: Node;
BEGIN
   IF tree.root = NIL THEN RETURN FALSE END;
   node := tree.root;
   WHILE node.left # NIL DO
      node := node.left;
   END;
   object := node.object;
   RETURN TRUE
END First;

PROCEDURE Last(tree: Tree;
               VAR object: Objects.Object) : BOOLEAN;
   (* returns last object in tree *)
   VAR
      node: Node;
BEGIN
   IF tree.root = NIL THEN RETURN FALSE END;
   node := tree.root;
   WHILE node.right # NIL DO
      node := node.right;
   END;
   object := node.object;
   RETURN TRUE
END Last;
```

Iterationen im sortierten binären Baum

SortedBinaryTrees.om

```
PROCEDURE Next(tree: Tree; key: Objects.Object;
               VAR object: Objects.Object) : BOOLEAN;
   (* returns lowest object in tree that is > key *)
   VAR
      node: Node;

   PROCEDURE NextNode(node: Node) : BOOLEAN;
   BEGIN
      IF node # NIL THEN
         IF tree.compare(key, node.object) < 0 THEN
            IF ~NextNode(node.left) THEN
               object := node.object;
            END;
            RETURN TRUE
         ELSE
            RETURN NextNode(node.right)
         END;
      END;
      RETURN FALSE
   END NextNode;

BEGIN (* Next *)
   ASSERT(tree.comparable(key));
   RETURN NextNode(tree.root)
END Next;
```

Iterationen im sortierten binären Baum

SortedBinaryTrees.od

```
TYPE
   VisitProc = PROCEDURE (object: Objects.Object);
      (* called by Traverse for all objects of a tree *)
```

SortedBinaryTrees.om

```
PROCEDURE Traverse(tree: Tree; visit: VisitProc);
   (* call visit for all objects in tree in sorted order *)

   PROCEDURE TraverseInorder(node: Node);
   BEGIN
      IF node # NIL THEN
         TraverseInorder(node.left);
         visit(node.object);
         TraverseInorder(node.right);
      END;
   END TraverseInorder;

BEGIN (* Traverse *)
   TraverseInorder(tree.root);
END Traverse;
```

Literaturhinweise zu Datenstrukturen auf Basis von Bäumen

In dieser Einführung fehlen leider viele wichtige und interessante Techniken und Varianten zu diesem Thema. Mehr hierzu gibt es bei:

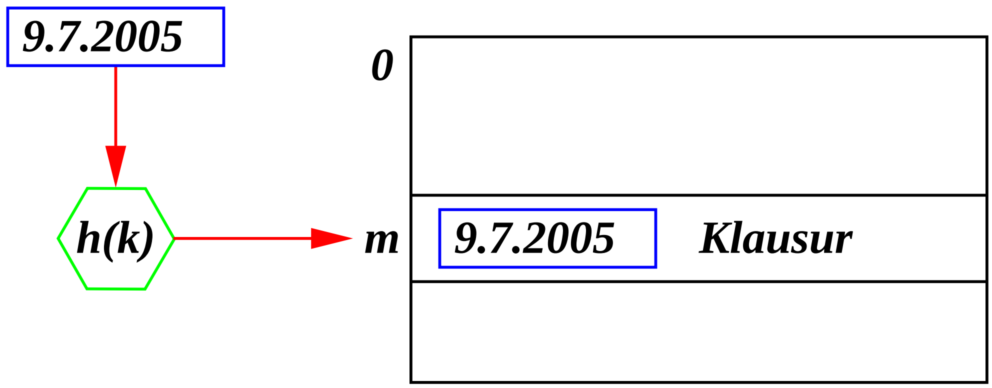
- Donald E. Knuth, “The Art of Computer Programming”, Band 1 und 3 mit den Kapiteln 2.3 (über Bäume), 6.2.2 (Suche in binären Bäumen) und 6.2.3 (ausgeglichene Bäume).

- Eine Implementierung höhenausgeglichener Bäume in Oberon findet sich in dem Modul *AVLTrees* von Sven Lutz im Verzeichnis
  `/home/obsrc/lib/collections/lib`.
  AVL steht übrigens für Adelson-Velsky und Landis, den beiden russischen Mathematikern, die 1962 die zugehörigen Algorithmen zuerst entdeckt hatten.

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, “Introduction to Algorithms”, Kapitel 13 ff.

- Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, “Data Structures and Algorithms”, Kapitel 3.

Hash-Verfahren



- Hash-Verfahren versuchen, die Ermittlung eines Datensatzes in Abhängigkeit eines Schlüssels $k \in K$ mit Hilfe einer Hash-Funktion $h : K \rightarrow M$ zu vereinfachen, die einen Schlüssel k in eine Speicherposition $m \in M = [0..n-1]$ abbildet.

- Leider ist es nicht trivial, selbst für eine auch nur relativ kleine vorgegebene Menge von Schlüsseln K eine Hash-Funktion zu finden, die injektiv ist, d.h.

$$\forall k_1, k_2 \in K : h(k_1) \neq h(k_2)$$

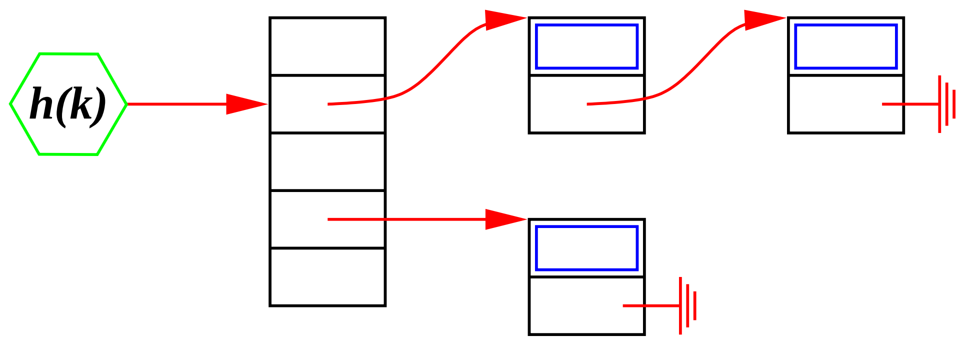
- Entsprechend sind Kollisionen auch bei geringen Mengen an Schlüsseln sehr wahrscheinlich (Geburtstagsproblem).

- Problemstellungen:

  - Wie wird mit Kollisionen umgegangen?

  - Wie könnte eine "gute" Hash-Funktion aussehen?

Kollisionen



- Die einfachste (und gängigste) Technik zur Lösung von Kollisionen besteht in der Bildung von linearen Listen. In der eigentlichen Tabelle (über den Indexbereich M) sind dann nur Zeiger, die dann entweder **NIL** sind (bislang noch kein k mit $h(k) = m$ gesehen) oder lineare Listen.

- Eine solche Tabelle wird *bucket table* genannt.

- Im einfachsten Falle werden neue Elemente bei der jeweiligen Liste ganz vorne eingefügt. Es ist aber auch denkbar, die Listen sortiert zu verwalten oder entsprechend der Zugriffshäufigkeit zu organisieren, um im Falle längerer Kollisionslisten die Zugriffszeit zu reduzieren.

- Es gibt auch zahlreiche Verfahren, bei denen die Datensätze direkt in der durch M indizierten Tabelle untergebracht werden, wo dann im Falle einer Kollision andere, noch freie, Plätze ausgesucht werden. Nachteile: Limitierte Anzahl von Datensätzen (oder die Notwendigkeit von Überlauftabellen), Suchzeit wird u.U. länger und das Löschen aufwendiger.

Hash-Funktionen

- Anforderungen an Hash-Funktionen:

  1. Effiziente Berechenbarkeit,

  2. Surjektivität (ansonsten wird Speicherplatz innerhalb von M verschwendet) und

  3. gute “Streuung”, d.h. auch relativ “ähnliche” Schlüssel sollten verschiedenen Lokationen aus M zugeordnet werden.

- Hash-Funktionen müssen in Abhängigkeit von K geeignet bestimmt werden.

- Kryptographische Hash-Funktionen wie MD5 und SHA-1 erfüllen zwar den 2. und 3. Punkt in hervorragender Weise für beliebige Schlüsselmengen K, sind jedoch leider nicht effizient genug.

- Wenn k eine ganze Zahl ist, gibt es zwei bewährte Varianten:

  - Division: $h(k) = k \bmod n$.

  - Multiplikation: $h(k) = \left\lfloor n \left(\left(\frac{A}{w} k \right) \bmod 1 \right) \right\rfloor$

  ($M = [0..n-1], w = \textbf{MAX}(\textbf{INTEGER}) + 1$, und A ist eine natürliche Zahl, die teilerfremd zu w ist)

Auf Division basierende Hash-Funktionen

$$h(k) = k \bmod n$$

- Die Streuung dieser Variante hängt von der Wahl von n ab.
- Wenn beispielsweise n gerade ist, dann wäre $h(k)$ genau dann gerade, wenn k gerade ist.
- Wenn n eine Zweierpotenz wäre, würden einfach die höherwertigen Bits von k abgeschnitten werden.

Auf Multiplikation basierende Hash-Funktionen

$$h(k) = \left\lfloor n \left(\left(\frac{A}{w} k \right) \bmod 1 \right) \right\rfloor$$

- Hierbei ist $w = \textbf{MAX}(\textbf{INTEGER}) + 1$ und und A eine natürliche Zahl, die teilerfremd zu w ist.

- Die binäre Repräsentierung von A kann dann als A/w betrachtet werden, wenn gedanklich links von A eine Null, gefolgt von einem Komma, steht.

- Die Multiplikation von A und k würde bei einer Wortbreite von 32 Bit insgesamt 64 Bit benötigen.

- Wenn jedoch n eine Zweier-Potenz 2^t ist, dann sind nur die führenden t Bits der niedrigwertigeren 32 Bit des Ergebnisses der Multiplikation interessant.

Auf Multiplikation basierende Hash-Funktionen

```
CONST bitsperword = 32; (* size of INTEGER in bits *)
TYPE HashValue = INTEGER;
   (* [0..n-1], n = 2^t, t < bitsperword *)

PROCEDURE Hash(k: INTEGER) : HashValue;
BEGIN
   RETURN SHORT(SYSTEM.LSH(a * k, t - bitsperword))
END Hash;
```

- In Oberon liefert die Multiplikation von zwei 32-Bit-Zahlen nur die niedrigwertigen 32 Bits.
- Die Operation **LSH** (*logical shift*) aus dem Modul **SYSTEM** verschiebt die Bits des 1. Parameters um die im 2. Parameter angegebene Anzahl. Da $t - bitsperword$ negativ ist, wird nach rechts geschoben.
- Die links nachkommenden Bits werden dabei alle auf 0 gesetzt.
- Entsprechend ergeben sich anschließend die t niedrigwertigen Bits aus der Multiplikation, und die verbleibenden Bits sind 0.
- Das Resultat liegt somit im Intervall $[0, 2^t - 1]$.

Auf Multiplikation basierende Hash-Funktionen

Satz von Hugo Steinhaus und Vera Turán Sós:

Sei θ eine irrationale Zahl. Wenn die Punkte $\{\theta\}, \{2\theta\}, ..., \{n\theta\}$ in das Intervall $[0, 1]$ plaziert werden (jeweils mod 1), dann haben die dadurch entstandenen $n + 1$ Teilintervalle von $[0, 1]$ maximal drei verschiedene Längen. Wenn der nächste Punkt $\{(n + 1)\theta\}$ hinzugefügt wird, dann fällt er in eines der längsten Segmente.

- Dieser Satz stellt sicher, daß diese Punkte sehr gleichmäßig über das Intervall von $[0, 1]$ verteilt werden und damit darauf basierende Hash-Funktionen eine gute Streuwirkung besitzen.

- Allerdings gibt es durchaus Unterschiede in der Wahl von θ, die sich in den möglichen Größenunterschieden der Intervalle bemerkbar machen. Wenn beispielsweise θ nahe bei 0 oder 1 gewählt wird, würde es zu Beginn viele winzige Teilintervalle und ein großes geben.

- Der goldene Schnitt $\phi^{-1} = (\sqrt{5} - 1)/2$ ist ein idealer Kandidat für θ, wenn eine möglichst gleichmäßige Verteilung gewünscht wird.

- A sollte dann als die nächstgelegene ganze Zahl zu $\phi^{-1}w$ gewählt werden, die teilerfremd zu w ist.

- Somit ist beispielsweise $A = 1327217883$ geeignet für $w = 2^{31}$.

Andere Schlüsseltypen

```
TYPE HashValue = INTEGER; (* 0..1023 *)

PROCEDURE HashString(string: ARRAY OF CHAR) : HashValue;
   VAR
      index: INTEGER;
      hashval, ordval: INTEGER;
      ch: CHAR;
BEGIN
   hashval := 0; index := 0;
   WHILE (index < LEN(string)) & (string[index] # 0X) DO
      ch := string[index];
      IF ch >= " " THEN
         ordval := ORD(ch) - ORD(" ");
      ELSE
         ordval := ORD(MAX(CHAR)) - ORD(" ") + ORD(ch);
      END;
      hashval := SHORT(SYSTEM.ROT(hashval, -5)) + ordval;
      INC(index);
   END;
   RETURN SHORT(SYSTEM.LSH(1327217883 * hashval, 10 - 32))
END HashString;
```

- In der Praxis kommen viele weitere Schlüsseltypen neben **INTEGER** vor, so daß zuvor eine Konvertierung stattfinden muß.

- Längere Schlüssel (insbesondere Zeichenketten) müssen dabei so geschickt zu einer **INTEGER** verdichtet werden, daß auch die Reihenfolge relevant ist, d.h. `"AB"` sollte einen anderen Hash-Wert haben als `"BA"`.

- Sehr beliebt ist es daher, Summation und Rotation zu verbinden.

Eine Abstraktion für assoziative Arrays

- Assoziative Arrays (andere Namen: *hashes* und *dictionaries*) sind in den Grundoperationen vergleichbar mit normalen Arrays, abgesehen davon, daß
  - der Index-Typ beliebig sein kann (z.B. Zeichenketten) und
  - nicht notwendigerweise eine vollständige Ordnung für den Index-Typ existiert.
- In vielen Anwendungen sind assoziative Arrays sehr praktisch. So ist es z.B. bei einer Adreßdatenbank sinnvoll, wenn der Name als Index-Typ dient. Häufig wird der Index auch Schlüssel genannt.
- Manchmal sind auch Zugriffe über mehrere Schlüssel interessant. Bei Adreßdatenbanken könnte es beispielsweise nützlich sein, von der Telefon-Nummer oder der E-Mail-Adresse auf einen Datensatz zu schließen. Das sind dann sekundäre Schlüssel, die nicht notwendigerweise bei jedem Datensatz definiert sind.
- Das Hash-Verfahren ist ideal, um assoziative Arrays zu implementieren.

Eine Abstraktion für assoziative Arrays

- Folgendes wird bei einem konkreten assoziativen Array in Abhängigkeit des Datentyps benötigt:

  - Eine Hash-Funktion. Damit sie von der Tabellengröße unabhängig ist, kann sie einfach einen Wert vom Typ **INTEGER** zurückliefern. Die Implementierung des assoziativen Arrays kann dann den Wert in das Intervall $[0, n-1]$ abbilden.

  - Eine Operation, die zwei Schlüssel auf Identität prüft. Da die Hash-Funktion typischerweise nicht injektiv ist, müssen Schlüssel mit dem gleichen Hash-Wert voneinander unterschieden werden können.

  - Wie zuvor bei den binären sortierten Bäumen ist es sinnvoll, noch eine Operation hinzuzufügen, die überprüft, ob ein Objekt zu einem konkreten assoziativen Array "passt".

- Analog zu *SortedBinaryTrees* gibt es dann die Operationen *Insert*, *Delete* und *Lookup*.

- Da es jedoch keine Sortierung gibt, sind Operationen wie *Next*, *Prev* und eine Traverse in sortierter Reihenfolge nicht möglich. Stattdessen kann nur eine Iteration mit einer undefinierten Reihenfolge unterstützt werden.

Eine Abstraktion für assoziative Arrays

Hashes.od

```
TYPE
   HashValue = INTEGER;

   HashValProc =
      PROCEDURE (object: Objects.Object) : HashValue;
      (* return hash value of the key of this object *)

   HashableProc =
      PROCEDURE (object: Objects.Object) : BOOLEAN;
      (* return TRUE if hashval and equal may be invoked
         for object *)

   EqualProc =
      PROCEDURE (object1, object2: Objects.Object) : BOOLEAN;
      (* return TRUE if the keys of object1 and object2
         are identical *)

TYPE
   Hash = POINTER TO HashRec;
   HashRec = RECORD (Objects.ObjectRec) END;

PROCEDURE Create(VAR hash: Hash;
                 hashable: HashableProc;
                 hashval: HashValProc;
                 equal: EqualProc);
```

- Genauso wie *SortedBinaryTrees.Create* erhält *Hashes.Create* alle Prozeduren als Parameter, die von dem Typ des Schlüssels abhängen.

Eine Abstraktion für assoziative Arrays

Hashes.od

```
PROCEDURE Hashable(hash: Hash;
                   object: Objects.Object) : BOOLEAN;
   (* return TRUE if the key of object is compatible
      to those of hash
   *)

PROCEDURE Acceptable(hash: Hash;
                     object: Objects.Object) : BOOLEAN;
   (* return TRUE if the key of object is hashable and
      not yet present in hash
   *)

PROCEDURE Insert(hash: Hash; object: Objects.Object);
   (* precondition: object must be acceptable for hash
      adds object to hash
   *)

PROCEDURE Delete(hash: Hash; key: Objects.Object);
   (* precondition: key must be hashable for hash
      deletes object out of hash that has a key that
      is considered equal to key
   *)

PROCEDURE Lookup(hash: Hash; key: Objects.Object;
                 VAR object: Objects.Object) : BOOLEAN;
   (* precondition: key must be hashable for hash;
      stores into object that object of the hash
         that is identical to key if present;
      returns TRUE if one object has been found
   *)
```

Eine Abstraktion für assoziative Arrays

Hashes.od

```
PROCEDURE IterateHash(hash: Hash);
   (* start iteration of hash *)

PROCEDURE Next(hash: Hash;
               VAR object: Objects.Object) : BOOLEAN;
   (* store next object of iteration into object;
      note that the order of objects is undefined;
      returns FALSE on end of iteration
   *)
```

- *Hashes.IterateHash* startet einen Durchgang und *Hashes.Next* liefert die im assoziativen Array enthaltenen Objekte in einer vom Hash-Wert abhängigen Reihenfolge.

AddressManager.om

```
Hashes.IterateHash(db.hashByName);
WHILE Hashes.Next(db.hashByName, address) DO
   Addresses.PrettyPrint(Streams.stdout, address);
END;
```

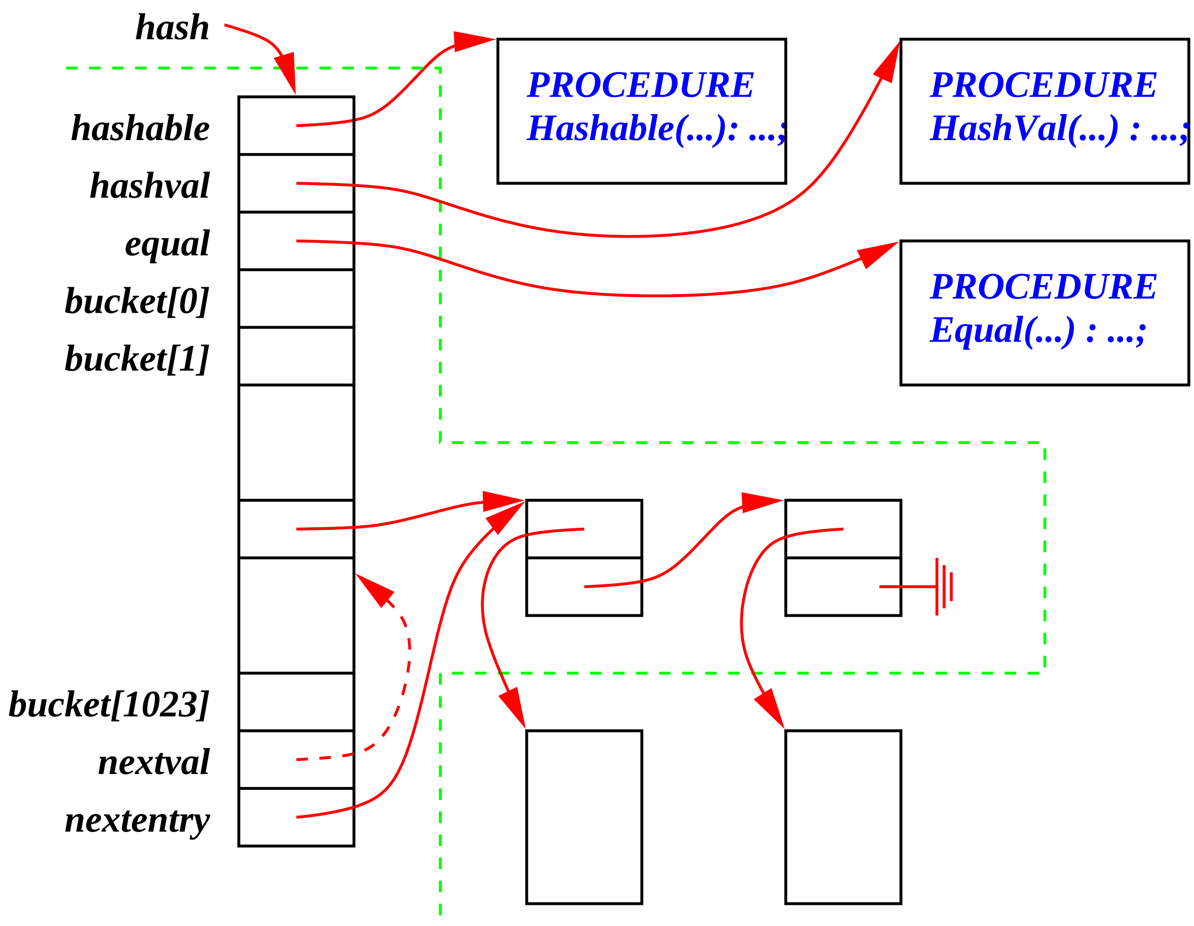
Implementierung assoziativer Arrays

Hashes.om

```
CONST
   tabsize = 1024; logtabsize = 10;
TYPE
   Entry = POINTER TO EntryRec;
   EntryRec =
      RECORD
         object: Objects.Object;
         next: Entry; (* next entry with same hash value *)
      END;
   BucketTable = ARRAY tabsize OF Entry;
TYPE
   Hash = POINTER TO HashRec;
   HashRec =
      RECORD
         (Objects.ObjectRec)
         hashable: HashableProc;
         hashval: HashValProc;
         equal: EqualProc;
         bucket: BucketTable;
         (* for iterations: *)
         nextval: HashValue; nextentry: Entry;
      END;
```

- *hashable*, *hashval* und *equal* sind die Prozeduren, die bei *Hashes.Create* übergeben worden sind.
- *bucket* ist die *bucket table*, die mit den Hash-Werten indiziert wird und jeweils auf die Liste der Objekte mit dem gleichen Hash-Wert verweist.

Implementierung assoziativer Arrays



- Die gestrichelte (grüne) Linie kennzeichnet die Privatsphäre von *Hashes*.

Implementierung assoziativer Arrays

Hashes.om

```
PROCEDURE HashVal(hash: Hash;
                  object: Objects.Object) : HashValue;
   (* converts provided hash value into a hash value inside
      our range of [0..tabsize-1]
   *)
BEGIN
   RETURN
      SHORT(SYSTEM.LSH(1327217883 * hash.hashval(object),
                       logtabsize - 32))
END HashVal;

PROCEDURE Find(hash: Hash; key: Objects.Object;
               VAR hashval: HashValue;
               VAR entry, predecessor: Entry) : BOOLEAN;
BEGIN
   hashval := HashVal(hash, key);
   entry := hash.bucket[hashval]; predecessor := NIL;
   WHILE (entry # NIL) & ~hash.equal(entry.object, key) DO
      predecessor := entry; entry := entry.next;
   END;
   RETURN entry # NIL
END Find;
```

- *HashVal* bildet einen **INTEGER**-Wert, den die typabhängige Hash-Funktion liefert, in den Indexbereich der Hash-Tabelle ab.
- *Find* sucht nach *key* innerhalb von *hash*. Diese Prozedur wird von *Acceptable*, *Insert*, *Delete* und *Lookup* verwendet.

Implementierung assoziativer Arrays

Hashes.om

```
PROCEDURE Create(VAR hash: Hash;
                 hashable: HashableProc;
                 hashval: HashValProc;
                 equal: EqualProc);
BEGIN
   NEW(hash);
   hash.hashable := hashable; hash.hashval := hashval;
   hash.equal := equal;
   hash.nextval := tabsize + 1; hash.nextentry := NIL;
END Create;

PROCEDURE Hashable(hash: Hash;
                   object: Objects.Object) : BOOLEAN;
   (* return TRUE if the key of object is compatible
      to those of hash
   *)
BEGIN
   RETURN hash.hashable(object)
END Hashable;

PROCEDURE Acceptable(hash: Hash;
                     object: Objects.Object) : BOOLEAN;
   (* return TRUE if the key of object is hashable and
      not yet present in hash
   *)
   VAR
      hashval: HashValue;
      entry, predecessor: Entry;
BEGIN
   RETURN ~Find(hash, object, hashval, entry, predecessor)
END Acceptable;
```

Implementierung assoziativer Arrays

Hashes.om

```
PROCEDURE Insert(hash: Hash; object: Objects.Object);
   (* precondition: object must be acceptable for hash
      adds object to hash
   *)
   VAR
      ok: BOOLEAN;
      hashval: HashValue;
      entry, predecessor: Entry;
BEGIN
   ok := ~Find(hash, object, hashval, entry, predecessor);
   ASSERT(ok);
   NEW(entry);
   entry.object := object;
   entry.next := hash.bucket[hashval];
   hash.bucket[hashval] := entry;
END Insert;
```

- Analog zu *SortedBinaryTrees* wird hier darauf bestanden, daß Schlüssel eindeutig sind.
- Genauso wie bei den Stapeln werden neu hinzukommende Einträge vor dem ersten Eintrag eingefügt.

Implementierung assoziativer Arrays

Hashes.om

```
PROCEDURE Delete(hash: Hash; key: Objects.Object);
   (* precondition: key must be hashable for hash
      deletes object out of hash that has a key that
      is considered equal to key
   *)
   VAR
      hashval: HashValue;
      entry, predecessor: Entry;
BEGIN
   IF Find(hash, key, hashval, entry, predecessor) THEN
      IF predecessor = NIL THEN
         hash.bucket[hashval] := entry.next;
      ELSE
         predecessor.next := entry.next;
      END;
   END;
END Delete;
```

- Das Löschen erfolgt analog zu einfachen linearen Listen, wobei *hashval* nur bestimmt, auf welcher Liste operiert wird.
- *predecessor* wird von *Find* auf den Vorgänger des zu löschenden Eintrags gesetzt. Dieser ist **NIL**, wenn der erste Eintrag zu löschen ist.

Implementierung assoziativer Arrays

Hashes.om

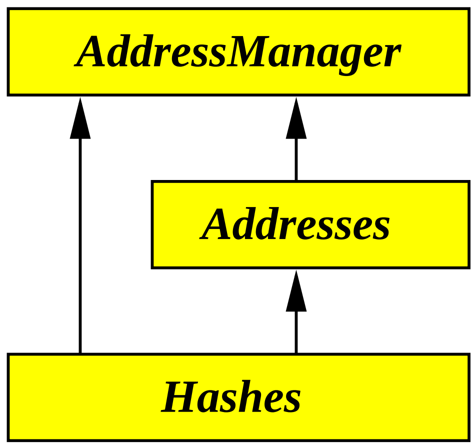
```
PROCEDURE Lookup(hash: Hash; key: Objects.Object;
                 VAR object: Objects.Object) : BOOLEAN;
   (* precondition: key must be hashable for hash;
      stores into object that object of the hash
         that is identical to key if present;
      returns TRUE if one object has been found
   *)
   VAR
      hashval: HashValue;
      entry, predecessor: Entry;
BEGIN
   IF Find(hash, key, hashval, entry, predecessor) THEN
      object := entry.object;
      RETURN TRUE
   ELSE
      RETURN FALSE
   END;
END Lookup;
```

Implementierung assoziativer Arrays

Hashes.om

```
PROCEDURE IterateHash(hash: Hash);
   (* start iteration of hash *)
BEGIN
   hash.nextval := 0; hash.nextentry := NIL;
END IterateHash;

PROCEDURE Next(hash: Hash;
               VAR object: Objects.Object) : BOOLEAN;
   (* store next object of iteration into object;
      note that the order of objects is undefined;
      returns FALSE on end of iteration
   *)
BEGIN
   IF hash.nextentry = NIL THEN
      WHILE (hash.nextval < tabsize) &
            (hash.bucket[hash.nextval] = NIL) DO
         INC(hash.nextval);
      END;
      IF hash.nextval >= tabsize THEN
         RETURN FALSE
      END;
      hash.nextentry := hash.bucket[hash.nextval];
      INC(hash.nextval);
   END;
   object := hash.nextentry.object;
   hash.nextentry := hash.nextentry.next;
   RETURN TRUE
END Next;
```

Anwendung: Eine kleine Adreßdatenbank



- Eine kleine Anwendung, die eine Adreßdatenbank realisiert, soll die Möglichkeiten von *Hashes* demonstrieren.

- Das Modul *Addresses* exportiert den Datentyp *Addresses.Address* zusammen mit zugehörigen Operationen zur Unterstützung von *Hashes* und zur Ein- und Ausgabe.

- Das Modul *AddressManager* offeriert eine interaktive Benutzerschnittstelle und dient als Hauptprogramm.

Der Datentyp *Addresses.Address*

Addresses.od

```
TYPE
   Address = POINTER TO AddressRec;
   AddressRec =
      RECORD
         (Objects.ObjectRec)
         name, email, address: ARRAY 128 OF CHAR;
         phone: ARRAY 32 OF CHAR;
      END;

PROCEDURE HashByName
            (object: Objects.Object) : Hashes.HashValue;
PROCEDURE HashByPhone
            (object: Objects.Object) : Hashes.HashValue;
PROCEDURE HashByEmail
            (object: Objects.Object) : Hashes.HashValue;

PROCEDURE EqualByName
            (object1, object2: Objects.Object) : BOOLEAN;
PROCEDURE EqualByPhone
            (object1, object2: Objects.Object) : BOOLEAN;
PROCEDURE EqualByEmail
            (object1, object2: Objects.Object) : BOOLEAN;

PROCEDURE Hashable(object: Objects.Object) : BOOLEAN;

PROCEDURE ReadFromFile(s: Streams.Stream;
                       VAR address: Address) : BOOLEAN;
PROCEDURE WriteToFile(s: Streams.Stream;
                      address: Address) : BOOLEAN;
PROCEDURE ReadFromUser(s: Streams.Stream;
                       VAR address: Address) : BOOLEAN;
PROCEDURE PrettyPrint(s: Streams.Stream; address: Address);
```

Der Datentyp *Addresses.Address*

Addresses.om

```
PROCEDURE HashByName
            (object: Objects.Object) : Hashes.HashValue;
BEGIN
   RETURN HashByString(object(Address).name)
END HashByName;

PROCEDURE HashByPhone
            (object: Objects.Object) : Hashes.HashValue;
BEGIN
   RETURN HashByString(object(Address).phone)
END HashByPhone;

PROCEDURE HashByEmail
            (object: Objects.Object) : Hashes.HashValue;
BEGIN
   RETURN HashByString(object(Address).email)
END HashByEmail;
```

- Für jede Art eines Zugriffs, sei es über den primären Schlüssel *name* oder einen der sekundären Schlüssel *phone* oder *email*, wird eine separate Hash-Funktion und eine separate Vergleichs-Operation benötigt.

Der Datentyp *Addresses.Address*

Addresses.om

```
PROCEDURE EqualByName
            (object1, object2: Objects.Object) : BOOLEAN;
BEGIN
   RETURN object1(Address).name = object2(Address).name
END EqualByName;

PROCEDURE EqualByPhone
            (object1, object2: Objects.Object) : BOOLEAN;
BEGIN
   RETURN object1(Address).phone = object2(Address).phone
END EqualByPhone;

PROCEDURE EqualByEmail
            (object1, object2: Objects.Object) : BOOLEAN;
BEGIN
   RETURN object1(Address).email = object2(Address).email
END EqualByEmail;

PROCEDURE Hashable(object: Objects.Object) : BOOLEAN;
BEGIN
   RETURN object IS Address
END Hashable;
```

Der Datentyp *Addresses.Address*

```
thales$ am
: insert
Name:    Hans Maier
Address: Ulm
Phone:   25902
E-Mail:  hans@maier.org
Inserted.
: insert
Name:    Elisabeth Huber
Address: Neu-Ulm
Phone:   94721
E-Mail:  eh@huber.de
Inserted.
: dump addresses.db
: quit
thales$ cat addresses.db
Hans Maier~hans@maier.org~Ulm~25902
Elisabeth Huber~eh@huber.de~Neu-Ulm~94721
thales$
```

- Einfache Datensätze können so in einer Datei abgelegt werden, daß
  - jede Zeile einem Datensatz entspricht und
  - die Felder durch Feldtrenner voneinander getrennt sind.
- In der Ulmer Oberon-Bibliothek ist die Definition eines Feldtrenners über das Modul *StreamDisciplines* möglich.

Der Datentyp *Addresses.Address*

Addresses.om

```
PROCEDURE ReadFromFile(s: Streams.Stream;
                       VAR address: Address) : BOOLEAN;
   VAR
      ok: BOOLEAN;
BEGIN
   NEW(address);
   ok :=
      Read.FieldS(s, address.name) &
      Read.FieldS(s, address.email) &
      Read.FieldS(s, address.address) &
      Read.FieldS(s, address.phone);
   Read.LnS(s);
   RETURN ok
END ReadFromFile;

PROCEDURE WriteToFile(s: Streams.Stream;
                      address: Address) : BOOLEAN;
   VAR
      fieldsep: CHAR;
BEGIN
   StreamDisciplines.GetFieldSep(s, fieldsep);
   Write.StringS(s, address.name); Write.CharS(s, fieldsep);
   Write.StringS(s, address.email); Write.CharS(s, fieldsep);
   Write.StringS(s, address.address);
   Write.CharS(s, fieldsep);
   Write.StringS(s, address.phone); Write.LnS(s);
   RETURN s.errors = 0
END WriteToFile;
```

Die Adreßdatenbank

```
thales$ am
: load addresses.db
: phone 25902
Name:    Hans Maier
Address: Ulm
Phone:   25902
E-Mail:  hans@maier.org
: email eh@huber.de
Name:    Elisabeth Huber
Address: Neu-Ulm
Phone:   94721
E-Mail:  eh@huber.de
: name "Hans Maier"
Name:    Hans Maier
Address: Ulm
Phone:   25902
E-Mail:  hans@maier.org
: quit
thales$
```

- Datensätze sollen aufgrund des Namens (primärer Schlüssel), der Telefon-Nummer oder der E-Mail-Adresse (sekundäre Schlüssel) gefunden werden.
- Entsprechend werden drei assoziative Arrays benötigt...

Die Adreßdatenbank

AddressManager.om

```
TYPE
   Database =
      RECORD
         hashByName, hashByPhone, hashByEmail: Hashes.Hash;
         (* names act as primary keys (i.e. we expect them
            to be unique) but we support phone numbers
            and email addresses as secondary indices as well;
            if secondary keys are not unique we just keep a
            reference to the first object we have seen with
            that key
         *)
      END;

PROCEDURE CreateDB(VAR db: Database);
   (* create the three hashes associated with db *)
BEGIN
   Hashes.Create(db.hashByName, Addresses.Hashable,
      Addresses.HashByName, Addresses.EqualByName);
   Hashes.Create(db.hashByPhone, Addresses.Hashable,
      Addresses.HashByPhone, Addresses.EqualByPhone);
   Hashes.Create(db.hashByEmail, Addresses.Hashable,
      Addresses.HashByEmail, Addresses.EqualByEmail);
END CreateDB;
```

- Für den primären und für die beiden sekundären Schlüssel wird je ein assoziatives Array unterhalten.

Die Adreßdatenbank

AddressManager.om

```
PROCEDURE Insert(db: Database;
                 address: Addresses.Address) : BOOLEAN;
   (* insert address into db (if it is acceptable,
      otherwise RETURN FALSE) by storing it into
      db.hashByName (accessible by the primary key) and to
      db.hashByPhone and hashByEmail if these fields are
      non-empty and these keys are not yet used
   *)
BEGIN
   IF Hashes.Acceptable(db.hashByName, address) THEN
      Hashes.Insert(db.hashByName, address);
      IF (address.phone # "") &
            Hashes.Acceptable(db.hashByPhone, address) THEN
         Hashes.Insert(db.hashByPhone, address);
      END;
      IF (address.email # "") &
            Hashes.Acceptable(db.hashByEmail, address) THEN
         Hashes.Insert(db.hashByEmail, address);
      END;
      RETURN TRUE
   ELSE
      RETURN FALSE
   END;
END Insert;
```

- Wenn ein neuer Eintrag hinzukommt, ist er in jeden der assoziativen Arrays einzufügen (soweit jeweils die Schlüssel gegeben sind).

- Da *Hashes* auf die Eindeutigkeit der Schlüssel besteht, kann pro sekundärem Schlüssel jeweils nur ein Eintrag erfolgen.

Die Adreßdatenbank

AddressManager.om

```
PROCEDURE Delete(db: Database; address: Addresses.Address);
   (* delete address from db.hashByName and
      the other two hashes if present
   *)
   VAR
      otherAddress: Addresses.Address;
BEGIN
   Hashes.Delete(db.hashByName, address);
   IF (address.phone # "") &
         Hashes.Lookup(db.hashByPhone,
            address, otherAddress) &
         (address = otherAddress) THEN
      Hashes.Delete(db.hashByPhone, address);
   END;
   IF (address.email # "") &
         Hashes.Lookup(db.hashByEmail,
            address, otherAddress) &
         (address = otherAddress) THEN
      Hashes.Delete(db.hashByEmail, address);
   END;
END Delete;
```

- Bei *Delete* sind ebenfalls alle assoziativen Arrays zu berücksichtigen.

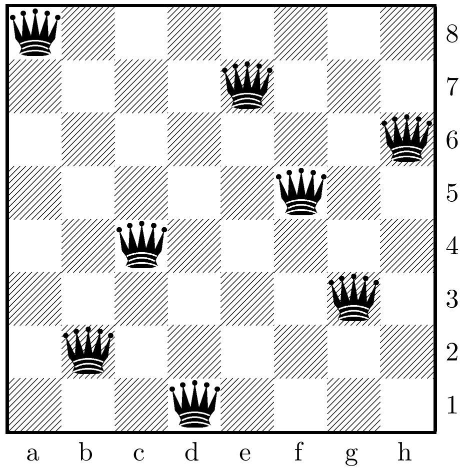
- Dabei ist hier zu achten, daß wir bei den sekundären Indizes nur die Einträge löschen, die von dem gleichen Datensatz stammen.

Literaturhinweise zu Datenstrukturen auf Basis von Hash-Verfahren

Zur weiteren Vertiefung sind folgende Texte geeignet:

- Donald E. Knuth, "The Art of Computer Programming", Band 3 "Sorting and Searching", Abschnitt 6.4 über "Hashing". Dieser Abschnitt trug wesentlich zu den einführenden Seiten dieses Kapitels bei.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, "Introduction to Algorithms", Kapitel 12 "Hash Tables".
- Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, "Data Structures and Algorithms", Kapitel 4 "Basic Operations on Sets" und speziell Abschnitt 4.7.
  Hier wird gezeigt, wie auf Basis von Hash-Tabellen eine Abstraktion für Mengen realisiert werden kann.

Backtracking



- Problem: Finde eine Stellung für n Damen auf einem Schachbrett der Größe $n \times n$, so daß sie sich nicht gegenseitig bedrohen.

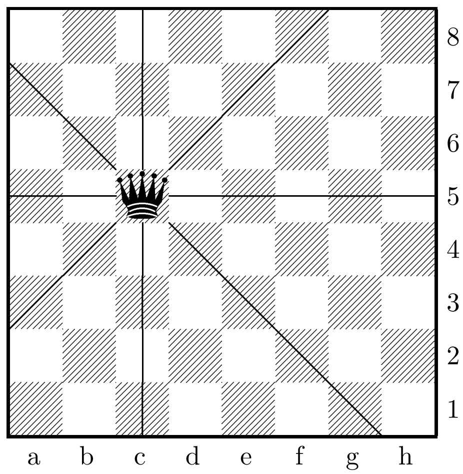
- Lösungsidee:

  - Schritt für Schritt eine Lösung aufbauen, indem eine Dame nach der anderen auf das Brett gestellt wird.

  - Wenn es keinen zulässigen Platz für die k-te Dame gibt, wird die $(k-1)$-te Dame vom Brett zurückgenommen und eine andere (noch nicht vorher probierte Position verwendet).

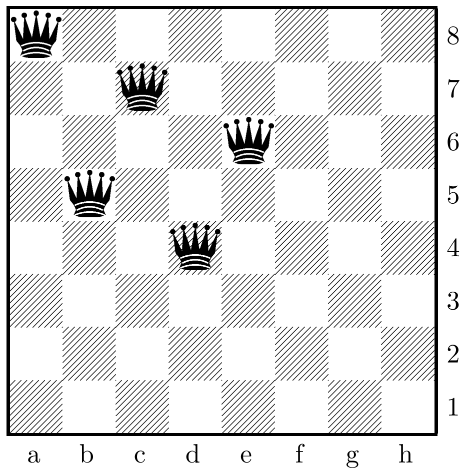
  - Dies wird solange durchprobiert, bis entweder eine Lösung gefunden wird oder alle Möglichkeiten durchprobiert sind und damit feststeht, daß es keine Lösung gibt.

Zugmöglichkeiten einer Dame



- Eine Dame im Schachspiel bedroht alle Felder
  - in der gleichen Zeile,
  - in der gleichen Spalte und
  - den beiden Diagonalen.

Sackgassen und Rückzüge



- Verfahren, die schrittweise Möglichkeiten durchprobieren und im Falle von "Sackgassen" zuvor gemachte Schritte wieder zurücknehmen, um neue Varianten zu probieren, nennen sich **Backtracking-Verfahren**.

- Eine Sackgasse beim 8-Damen-Problem zeigt das obige Diagramm, da es nicht möglich ist, eine weitere Dame unterzubringen.
  Hinweis: Hier wurde versucht, in jedem Schritt eine weitere Dame in der nächsten Zeile unterzubringen, beginnend mit der obersten (Zeile 8) und dann weiter nach unten fortfahrend.

- Um zu einer Lösung zu gelangen, muß hierbei nicht nur die 5. Dame zurückgenommen werden, sondern sogar alle Damen mit Ausnahme der allerersten.

- Erst wenn die 2. Dame von C7 auf E7 (oder auf F7 oder G7) vorgeschoben wird, gibt es eine Lösung.

Rahmen eines Backtracking-Verfahrens

```
PROCEDURE Solve(start: BOOLEAN;
                state: State; move: Move;
                VAR solution: State) : BOOLEAN;
   VAR
      moves: SetOfMoves;
BEGIN
   IF start THEN
      InitState(state);
   ELSE
      ApplyMove(state, move);
   END;
   IF Solved(state) THEN
      solution := state; RETURN TRUE
   END;
   moves := SetOfPossibleMoves(state);
   WHILE GetNextMove(moves, move) DO
      IF Solve(FALSE, state, move, solution) THEN
         RETURN TRUE
      END;
   END;
   RETURN FALSE
END Solve;
```

- Auch wenn das Backtracking-Verfahren auf verschiedene Problemfelder angewendet wird, so gleicht sich doch die Struktur der rekursiven Prozedur, die zur Lösungssuche verwendet wird.

- Viele Verfahren hängen jedoch davon ab, daß möglichst frühzeitig Irrwege erkannt werden, um mit einem vertretbaren Rechenzeitaufwand zur Lösung zu gelangen.

Reduzierung des Lösungsraumes

- Backtracking ist eine systematische Suche des gesamten Zustandsraumes nach akzeptablen Zuständen.

- In vielen Fällen ist der gesamte Zustandsraum zu umfangreich, um in akzeptabler Zeit vollständig untersucht zu werden. So gibt es beim n-Damen-Problem insgesamt $\frac{n^2!}{(n^2-n)!}$ Möglichkeiten, n Damen auf einem Schachbrett unterzubringen. Im Falle von $n = 8$ sind dies nicht weniger als $\frac{64!}{56!} = 178.462.987.637.760 > 10^{14}$ Möglichkeiten.

- Bei der schrittweisen Vorgehensweise des Backtracking liegt es daher nahe,

  - die Auswahl der möglichen Züge geschickt zu begrenzen und

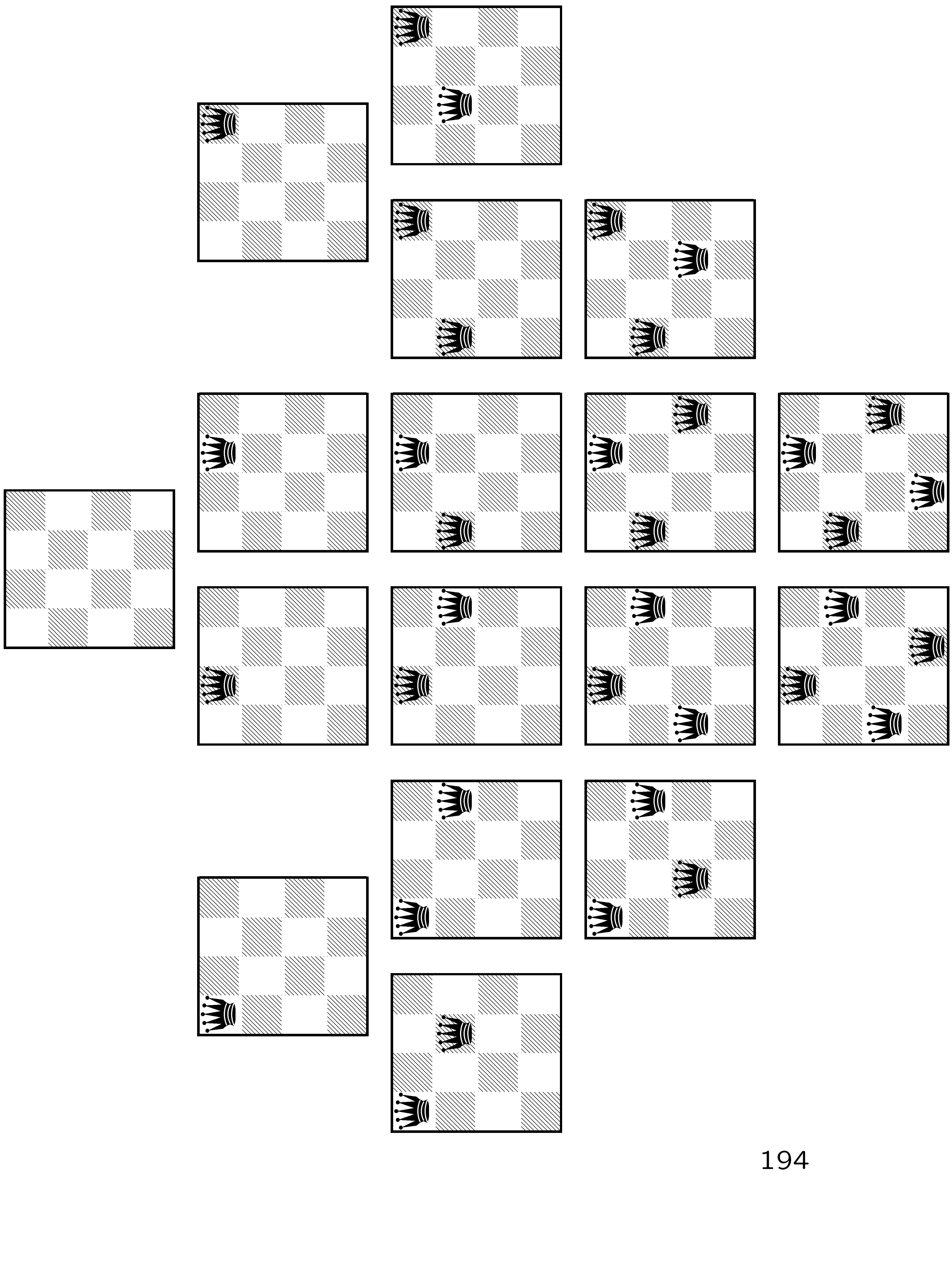
  - sofort zu überprüfen, ob von der Teil-Lösung gewisse notwendige Kriterien eingehalten werden, die sich aus der Bedingung für eine akzeptable Gesamt-Lösung ableiten lassen.

- Beim n-Damen-Problem ist es deswegen sinnvoll,

  - bei der k-ten Dame nur die k-te Zeile auf dem Schachbrett für die nächste Position zu berücksichtigen und

  - jede neu hinzukommende Dame sofort dahingehend zu untersuchen, ob sie bereits von den zuvor gesetzten Damen bedroht wird.

Pfade bei 4 Damen



Pfade bei 4 Damen

- Es gibt insgesamt $\frac{16!}{12!} = 43.680$ Möglichkeiten, 4 Damen auf einem 4×4-Brett unterzubringen.

- Wenn beim Setzen der k-ten Dame nur die k-te Zeile berücksichtigt wird, reduziert sich die Zahl auf n^n Möglichkeiten. Hier sind dies $4^4 = 256$.

- Wenn zudem sofort die Positionen verworfen werden, bei denen sich schon weniger als n Damen bedrohen, sind für $n = 4$ insgesamt 17 Zustände zu untersuchen, worunter 2 Lösungen und 4 Sackgassen sind.

Rahmen beim n-Damen-Problem

Queens.om

```
PROCEDURE Solve(start: BOOLEAN; board: Board;
                row, col: Position);
BEGIN
   IF ~start THEN
      AddQueen(board, row, col);
   END;
   IF board.nofqueens = board.size THEN
      WriteBoard(board); (* board is solved *)
   ELSE
      row := board.nofqueens;
      col := 0;
      WHILE col < board.size DO
         IF ~Threatened(board, row, col) THEN
            Solve(FALSE, board, row, col);
         END;
         INC(col);
      END;
   END;
END Solve;
```

- Abgesehen davon, daß hier alle Lösungen erwünscht sind, entspricht dies dem vorgestellten Rahmen.
- *row* und *col* spezifizieren jeweils die Position für die nächste Dame.
- Bei der k-ten Dame wird nur die k-te Zeile berücksichtigt.

O-Notation

- Generell ist es sehr sinnvoll, den Rechenzeitaufwand eines Algorithmus in Abhängigkeit von n (Datenumfang oder andere entscheidende Problemgröße) abzuschätzen.

- Hierzu eignet sich die von Paul Bachmann 1894 eingeführte O-Notation an:
  $g(n) = O(f(n))$ für $n \in \mathbb{N}$, falls $\exists M, n_0 \in \mathbb{N}$, so daß gilt
  $| \, g(n) \, | \leq | \, Mf(n) \, | \quad \forall \, n \geq n_0$.

- Einige Beispiele:

$O(1)$	konstanter Aufwand, unabhängig von n
$O(n)$	linearer Aufwand (z.B. Einlesen von n Zahlen)
$O(n \ln n)$	Aufwand guter Sortierverfahren
$O(n^2)$	quadratischer Aufwand
$O(n^k)$	polynomialer Aufwand (bei festem k)
$O(2^n)$	exponentieller Aufwand
$O(n!)$	Bestimmung aller Permutationen von n Elementen

- Die O-Notation hilft insbesondere bei der Beurteilung, ob ein Algorithmus für großes n noch geeignet ist bzw. erlaubt einen Effizienz-Vergleich zwischen verschiedenen Algorithmen für große n.

Wahl der Datenstruktur

- Mit entscheidend bei der Effizienz eines Backtracking-Algorithmus ist die Wahl einer geeigneten Datenstruktur.

- Wie effizient ist

  - die Überprüfung, ob eine Lösung erreicht worden ist,

  - die Bestimmung möglicher Schritte in einer vorgegebenen Situation,

  - die Überprüfung der partiellen Brauchbarkeit von Teil-Lösungen und

  - das Durchführen eines Schrittes?

- Nach Möglichkeit sollten all diese Schritte mit einem Aufwand von $O(1)$ durchführbar sein.

Wahl der Datenstruktur

```
CONST free = 0; queen = 1;
TYPE Piece = SHORTINT; (* free or queen *)
TYPE Position = SHORTINT; (* [0..n-1] *)
TYPE Board =
   RECORD
      nofqueens: SHORTINT;
      size: SHORTINT;
      piece: ARRAY maxn, maxn OF Piece;
   END;

PROCEDURE Threatened(board: Board;
                     row, col: Position) : BOOLEAN;
   (* check if a queen may be set at (row,col) without
      being in conflict with formerly set pieces
   *)
   VAR
      r, c: Position;
BEGIN
   (* check row *)
   c := 0;
   WHILE c < board.size DO
      IF board.piece[row, c] # free THEN RETURN FALSE END;
      INC(c);
   END;
   (* check column and both diagonals *)
   (* ... *)
   RETURN TRUE
END Threatened;
```

- Wenn als Datenstruktur ein 2-dimensionales Feld entsprechend dem Schachbrett verwendet wird, werden die Überprüfungen sehr aufwendig.

- 4 Schleifen jeweils mit n Schritten führen zu einem Aufwand von $O(n)$.

Wahl der Datenstruktur

```
TYPE Position = SHORTINT; (* [0..n-1] *)
TYPE Board =
   RECORD
      nofqueens: SHORTINT;
      size: SHORTINT;
      queens: ARRAY maxn OF RECORD row, col: Position END;
   END;

PROCEDURE Threatened(board: Board;
                     row, col: Position) : BOOLEAN;
   VAR
      index: SHORTINT;
      r, c: Position;
BEGIN
   index := 0;
   WHILE index < board.nofqueens DO
      r := board.queens[index].row;
      c := board.queens[index].col;
      IF (row = r) OR (col = c) OR
            (row + col = r + c) OR
            (ABS(row - col) = ABS(r - c)) THEN
         RETURN FALSE
      END;
      INC(index);
   END;
   RETURN TRUE
END Threatened;
```

- Wenn statt einem 2-dimensionalen Feld nur die Positionen der bislang gesetzten Damen vermerkt werden, vereinfacht sich die Überprüfung.

- Der Aufwand von *Threatened* reduziert sich jetzt auf eine Schleife mit $k-1$ Schritten für das Setzen der k-ten Dame.

Wahl der Datenstruktur

Queens.om

```
TYPE
   BoardSize = SHORTINT; (* [1..maxn] *)
   Position = SHORTINT; (* [0..n-1] *)
   Board =
      RECORD
         size: BoardSize; (* square size *)
         nofqueens: SHORTINT; (* [0..n], # of queens set *)
         pos: ARRAY maxn OF RECORD row, col: Position END;
            (* queen positions for [0..nofqueens-1] *)
         (* a queen on (row, col) threatens a row, a column,
            and 2 diagonals;
            rows and columns are characterized by
               their number (0..n-1),
            the diagonals by row-col+n-1 and row+col,
            (n is a shorthand for board.size)
         *)
         rows: SET; (* OF [0..n-1] *)
         cols: SET; (* OF [0..n-1] *)
         diags1: SET;
            (* OF [0..2*(n-1)] -- used for row-col+n-1 *)
         diags2: SET;
            (* OF [0..2*(n-1)] -- used for row+col *)
      END;
```

- Die nächste Verbesserung ist möglich, wenn die Mengen der bedrohten Zeilen, Spalten und Diagonalen mit verwaltet wird.

Wahl der Datenstruktur

Queens.om

```
PROCEDURE Threatened(board: Board;
                     row, col: Position) : BOOLEAN;
BEGIN
   ASSERT((row >= 0) & (row < board.size) &
          (col >= 0) & (col < board.size));
   RETURN (row IN board.rows) OR
          (col IN board.cols) OR
          (row - col + board.size - 1 IN board.diags1) OR
          (row + col IN board.diags2)
END Threatened;
```

- Der Überprüfungsaufwand von *Threatened* liegt jetzt bei $O(1)$.
- **ASSERT** erhält einen Ausdruck vom Typ **BOOLEAN** und führt zu einem Laufzeitfehler, falls dieser Ausdruck **FALSE** sein sollte. Auf diese Weise lassen sich recht elegant zusätzliche Überprüfungen einfügen.

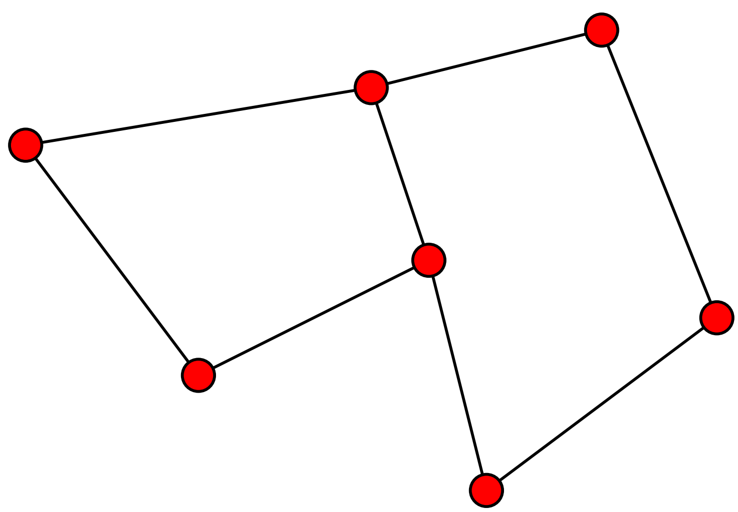
Wahl der Datenstruktur

Queens.om

```
PROCEDURE InitBoard(VAR board: Board; n: BoardSize);
BEGIN
   ASSERT((n > 0) & (n <= maxn));
   board.size := n;
   board.nofqueens := 0;
   board.rows := {};
   board.cols := {};
   board.diags1 := {};
   board.diags2 := {};
END InitBoard;

PROCEDURE AddQueen(VAR board: Board; row, col: Position);
BEGIN
   ASSERT(~Threatened(board, row, col));
   INCL(board.rows, row);
   INCL(board.cols, col);
   INCL(board.diags1, row - col + board.size - 1);
   INCL(board.diags2, row + col);
   board.pos[board.nofqueens].row := row;
   board.pos[board.nofqueens].col := col;
   INC(board.nofqueens);
END AddQueen;
```

- Auch das Hinzufügen einer Dame geschieht mit einem Aufwand von $O(1)$.

Travelling Salesman



- Gegeben sind n Orte und eine $n \times n$ Entfernungsmatrix M, wobei $M_{i,j}$ die Entfernung zwischen den Orten i und j angibt.

- Gesucht wird eine Reiseroute minimaler Länge, die alle Orte je genau einmal erreicht und zum Ausgangspunkt wieder zurückkehrt.

Branch and Bound

TSM.om

```
PROCEDURE Solve(VAR (* read-only *) problem: Problem;
                state: State; city: City;
                VAR solution: State);
BEGIN
   Travel(problem, state, city);
   IF state.nofvisits = problem.nofcities THEN
      IF MinimalSoFar(problem, state, solution) THEN
         solution := state;
      END;
   ELSIF (solution.minlen = -1) OR
      (state.length + state.minleft < solution.minlen) THEN
      city := 0;
      WHILE city < problem.nofcities DO
         IF ~(city IN state.visited) THEN
            Solve(problem, state, city, solution);
         END;
         INC(city);
      END;
   END;
END Solve;
```

- Wenn nach einem Optimum oder einer Verbesserung gesucht wird, lassen sich viele Zweige abschneiden, die nur noch schlechtere Lösungen produzieren können.
- Diese Technik nennt sich **Branch-And-Bound**-Verfahren.

Branch and Bound

- Sei ein Lösungsraum L und eine Kostenfunktion $c : L \to \mathbb{R}$ gegeben, wobei eine Lösung l_0 zu finden ist

  - mit $c(l_0) \leq k$ für eine vorgegebene Schranke k oder

  - $c(l_0) <= c(l) \, \forall \, l \in L$ (globales Minimum)

  dann ist es sinnvoll, bei einem Backtracking-Verfahren nach einer weiteren Kostenfunktion $c' : L' \to \mathbb{R}$ zu suchen, die für Teil-Lösungen l' aus L' eine "möglichst gute" untere Schranke gibt für alle $c(l)$ mit $l \in L$, wobei sich l von l' ableiten läßt.

- Die Kostenfunktion c' kann dann eingesetzt werden, um zu entscheiden, ob sich das Weiterverfolgen einer Teil-Lösung l' lohnt, oder ob es besser ist, diesen Zweig abzuschneiden.

- Beim Travelling-Salesman-Problem ist L die Menge aller Permutationen über die Orte $1..n$ und die Kostenfunktion die Länge einer Reiseroute. Die Kostenfunktion für Teil-Lösungen kann die bislang zurückgelegte Distanz berücksichtigen und eine untere Abschätzung über die noch zurückzulegende Strecke zu den noch nicht aufgesuchten Orten.

Rechenaufwand beim Problem des Travelling Salesman

- Wenn einfach nur alle Permutationen durchprobiert werden, liegt der Aufwand bei $O(n!)$ (bzw. $O((n-1)!)$ wenn berücksichtigt wird, daß es unerheblich ist, wo die Rundreise begonnen wird und somit der 1. Ort als Startort fest angenommen werden kann).

- Das Branch-And-Bound-Verfahren hilft in der Praxis, den Aufwand zu reduzieren, kann dies jedoch nicht in jedem Fall garantieren.

- Leider ist kein Algorithmus mit einer Komplexität von $O(n^k)$ (mit festem k) für dieses Problem bekannt. Allerdings gibt es auch bislang keinen Nachweis, daß es einen solchen Algorithmus nicht geben könnte.

- Somit ist dieses Problem heute z.B. für $n = 100$ nicht lösbar. Stattdessen gibt man sich dann mit relativ guten (aber nicht immer optimalen) Routen zufrieden.

- Dieses Problem gehört zu einer Klasse von Problemen (sogenannte **NP-vollständige** Probleme), die diese Ungewißheit teilen und für die gilt, daß wenn eines davon in polynomialer Zeit lösbar ist, dann sind auch alle anderen davon in polynomialer Zeit lösbar.

Komplexitätsklassen P und NP

- P ist die Menge aller Probleme, die sich in polynomialer Zeit auf einer Mehrband-Turing-Maschine (ist den uns vertrauten Computern äquivalent) lösen lassen.

- NP ist die Menge aller Probleme, die sich in polynomialer Zeit auf einer nicht-deterministischen Turing-Maschine (beliebige viele Pfade lassen sich parallel, jedoch ohne Kommunikation untereinander betrachten) lösen lassen.

- Bislang ist unbekannt, ob $P = NP$ oder $P \underset{\neq}{\subset} NP$.

- Die NP-vollständigen Probleme

  - gehören zu NP,

  - und falls eines von ihnen zu P gehören würde, dann gilt $NP = P$.

- Diese Theorie wurde von S. A. Cook (1971) und R. Karp (1972) entwickelt.