1.2.6 Beispiel: Fakultät

Wir wollen nun noch am Beispiel der Fakultät zeigen, daß in manchen Fällen auch ein rekursiver Algorithmus, abgesehen vom Overhead für die Funktionsaufrufe und dem Speicherbedarf für die Versorgungsblöcke, keine Nachteile bzgl. Laufzeit-Komplexität hat.

Definition der Fakultät

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ n \cdot (n-1)! & \text{für } n > 0 \end{cases}$$

Für die Laufzeit $T(n)$ des entsprechenden rekursiven Programms ergibt sich mit geeigneten Konstanten $a, b > 0$:

$$\begin{aligned} T(n) &= \begin{cases} a & n = 0 \\ b + T(n-1) & n > 0 \end{cases} \\ T(n) &= b + T(n-1) \\ &= 2 \cdot b + T(n-2) \\ &= \ldots \\ &= n \cdot b + T(0) \\ &= n \cdot b + a \\ &\in \Theta(n) \end{aligned}$$

Der Algorithmus hat eine Laufzeit $T(n)$, die linear in n ist, und hat damit (bis auf einen konstanten Faktor) das gleiche Laufzeitverhalten wie eine entsprechende iterative Implementierung.

1.3 Datenstrukturen

1.3.1 Datentypen

Die in Modula-3-ähnlichen Programmiersprachen verfügbaren Datentypen lassen sich in drei Gruppen unterteilen: elementare und zusammengesetzte Typen sowie Zeiger.

Elementare (atomare) Datentypen

- `INTEGER` : Ganze Zahlen
- `CARDINAL` : Natürliche Zahlen
- `CHARACTER` : Zeichen (Buchstaben etc.)
- `REAL` : Dezimalzahlen
- `BOOLEAN` : Wahrheitswerte (`TRUE` oder `FALSE`)

- zusätzlich existieren in Pascal und Modula-3 noch
  - Aufzählungstypen
    z.B. `TYPE Wochentag = {Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag} ;`
  - Unterbereichstypen
    z.B. `TYPE Lottozahl = [1..49] ;`

Operationen auf diesen Typen werden in der Regel durch elementare Hardware-Instruktionen realisiert.

Zusammengesetze Typen

Mit folgenden Typkonstruktoren lassen sich aus elementaren Datentypen zusammengesetzte Typen (compound types) bilden.

- `ARRAY` : Feld
- `RECORD` : Datensatz, Verbund, Struktur
- `SET` : Menge

Diese Typen erlauben direkten Zugriff auf ihre Komponenten, d.h. der Aufwand ist konstant, und nicht, wie z.B. bei verketteten Listen (siehe Abs. 1.3.2), von der Größe der Struktur abhängig.

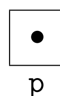
Zeigertypen (Pointer)

Der Datentyp *Zeiger* erlaubt *dynamische Datenstrukturen* zu erzeugen, d.h. Datenstrukturen deren Speicher erst bei Bedarf und zur Laufzeit angefordert wird. Unter einem *Zeiger* versteht man eine Variable, deren Wert als *Speicheradresse* einer anderen Variable verwendet wird. Man sagt, diese Variable werde durch den Zeiger *referenziert*, bzw. der Zeiger sei eine *Referenz* auf diese Variable. Um über den Zeiger auf die *referenzierte* Variable zuzugreifen, muss dieser *dereferenziert* werden. Die referenzierte Variable ist eine Variable vom *Bezugstyp*, der Zeiger ist eine Variable vom *Referenztyp*.

Erzeugung eines Pointers

Eine Zeigervariable wird angelegt, wenn eine Variable als Zeiger auf einen Bezugstyp deklariert wird. In Modula-3 dient dazu das Schlüsselwort `REF`.

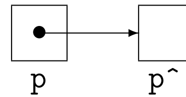
```
TYPE ZeigerTyp = REF BezugsTyp ;
VAR p : ZeigerTyp ;
```


p

Erzeugung einer Variablen vom Bezugstyp

Die Funktion `NEW` alloziert dynamisch, d.h. zur Laufzeit, Speicherplatz für eine Variable vom Bezugstyp und liefert die entsprechende Adresse zurück. Der Wert der referenzierten Variablen `p^` ist noch unbestimmt.

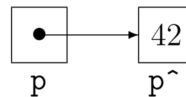
```
p := NEW(ZeigerTyp) ;
```



Dereferenzierung

Zugriff auf die referenzierte Variable erfolgt mit dem Dereferenzierungs-Operator `^`:

```
p^ := 42 ;
```



Zuweisungen an einen Pointer

`p := q ;`	`p` und `q` müssen denselben Bezugstyp haben.
`p := NIL ;`	Die Konstante `NIL` bedeutet, daß ein Zeiger auf keine Variable zeigt.

Freigabe einer Variablen vom Bezugstyp

In Modula-3 wird eine mit `NEW` erzeugte Variable automatisch entfernt, wenn kein Zeiger mehr auf sie verweist (*gargabe collection*). In vielen andere Sprachen (Pascal, Modula-2, C) muss man den belegten Speicher dagegen explizit freigeben. In Pascal geschieht dies z.B. mit der Prozedur `dispose`.

Benutzerdefinierte Datentypen

Mit Hilfe der Konstruktoren `ARRAY` und `RECORD` sowie der Zeiger lassen sich weitere, komplexere Datentypen konstruieren. Einige wichtige, die wir in den nächsten Abschnitten besprechen werden, sind die folgenden:

list: Liste, Sequenz, Folge

stack: Stapel, Keller, LIFO-Memory (last-in, first-out)

queue: Schlange, FIFO-Memory (first-in, first-out),

tree: Baum

Die genannten Datenstrukturen werden in der Regel dazu verwendet, Elemente eines bestimmten Typs zu speichern. Man spricht daher auch von *Container-Typen*. Den Typ

des enthaltenen Elements wollen wir nicht festlegen und verwenden in den folgenden Beispielen die Bezeichnung `ItemType`, die mittels einer Deklaration folgenden Typs mit Inhalt zu füllen ist:

```
TYPE ItemType = ... ;
```

Die Beispielprogramme wurden zum großen Teil [Sedgewick 93] entnommen.

1.3.2 Listen

Eine Liste stellt eine Sequenz oder Folge von Elementen $a_1, \ldots, a_n$ dar. Listen werden oft als Verkettung von Records mittels Zeigern (*verkettete Liste, linked list*) implementiert. Wenn eine Liste nicht verzweigt ist, nennt man sie auch *lineare* Liste. Die folgenden Operationen sind auf einer Liste definiert:

insert : Einfügen an beliebiger Position.

delete : Entfernen an beliebiger Position.

read : Zugriff auf beliebige Position.

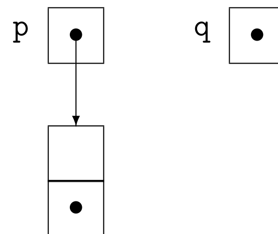
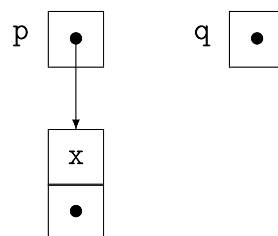
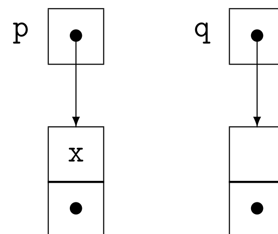
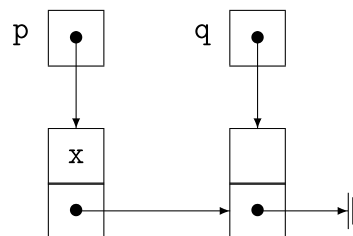
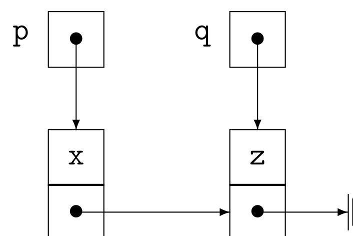
Bei der Implementierung kann man je nach Zielsetzung mit Pointern oder mit Arrays arbeiten.

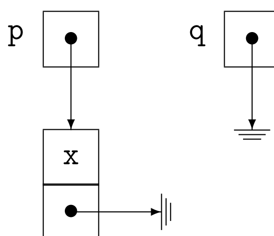
Zeiger-Implementierung von Listen (verkettete Liste)

In der Pointer-Implementierung wird eine Liste aus mehreren *Knoten* aufgebaut. Jeder Knoten enthält den Inhalt des Listenelements (key) und einen Zeiger auf den nachfolgenden Knoten (next).

```
TYPE link = REF node ;
     node = RECORD
         key  : ItemType ;
         next : link ;
     END ;
```

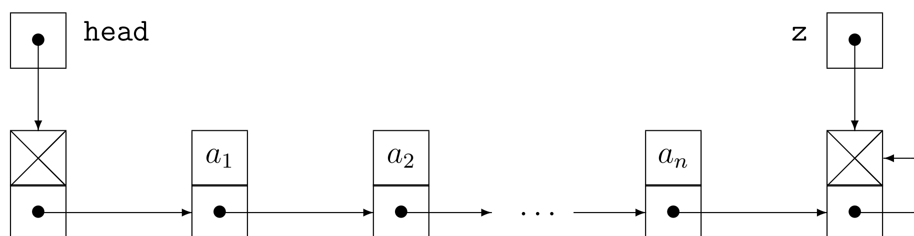
Beispiel:

Anweisung	**Wirkung**
`VAR p, q : link ;`	
`p := NEW(link) ;`	
`p^.key := x ;`	
`q := NEW(link) ;`	
`p^.next := q ;` `q^.next := NIL ;`	
`q^.key := z ;`	

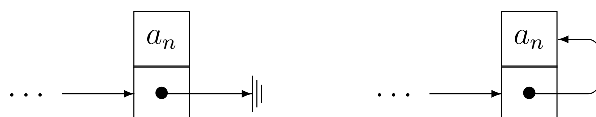
Anweisung	**Wirkung**
`p^.next := NIL ;` `q := NIL ;`	

Schematische Darstellung

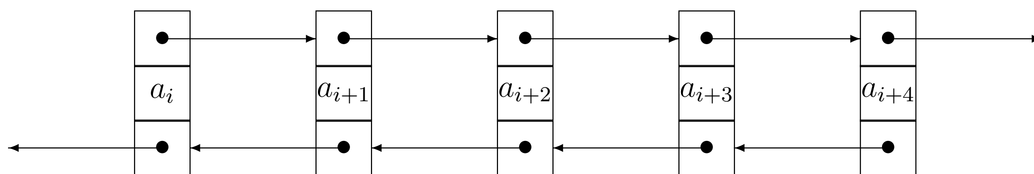
Eine einfach verkettete Liste sieht so aus:



Bei der Darstellung des Anfangs und Endes der Liste gibt es einige Varianten. In der hier vorgestellten Variante werden zwei zusätzliche Knoten `head` und `z` verwendet, die Anfang bzw. Ende der Liste repräsentieren und selbst kein Listenelement speichern. Andere Implementierungen verzichten auf ein explizites Kopf-Element. Beim Listenende sind auch folgende Varianten üblich:



Eine *doppelt verkettete* Liste lässt sich so darstellen:



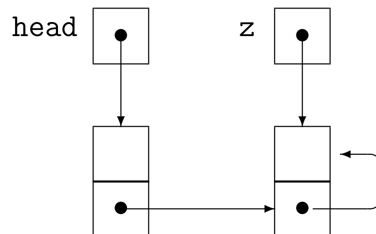
Andere Varianten sind *zyklische Listen* und *Skip-Listen*.

Programme

```
VAR head, z: link ;
```

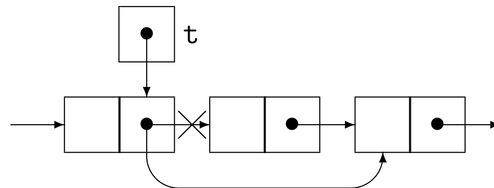
leere Liste anlegen

```
PROCEDURE ListInitialize() =
BEGIN
    head := NEW(link) ;
    z    := NEW(link) ;
    head^.next := z ;
    z^.next    := z ;
END ListInitialize ;
```



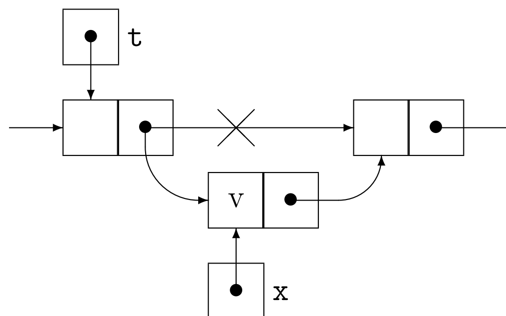
Nachfolger von t^ löschen

```
PROCEDURE DeleteNext(t : link) =
BEGIN
    t^.next := t^.next^.next ;
END DeleteNext ;
```



Nachfolger von t^ einfügen

```
PROCEDURE InsertAfter(v : ItemType ;
                      t : link) =
VAR x : link ;
BEGIN
    x := NEW(link) ;
    x^.key := v ;
    x^.next := t^.next ;
    t^.next := x ;
END InsertAfter ;
```



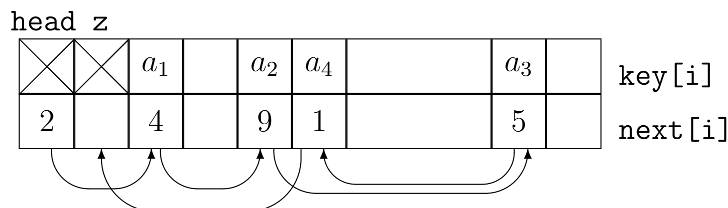
Array-Implementierung von Listen

Zwei auf Arrays basierende Implementierungen der linearen Liste sind das sequentielle Array und die Cursor-Darstellung.

Sequentielles Array

Cursor-Darstellung

Die Cursor-Darstellung ist an die Zeiger-Implementierung angelehnt und verwendet ein zweites Array für die Indizes der Nachfolgeelemente.



Die Listenmethoden `InsertAfter` und `DeleteNext` lassen sich in der Cursor-Darstellung wie folgt implementieren. Dabei ist `z` wieder das Listenende; `x` bezeichnet die nächste freie Position im Array (sehr primitive Speicherverwaltung).

```
VAR key  : ARRAY [0..N] OF ItemType ;
    next : ARRAY [0..N] OF CARDINAL ;
    x, head, z : CARDINAL ;

PROCEDURE ListInitialize() =
BEGIN
    head := 0 ; z := 1 ; x := 1 ;
    next[head] := z ; next[z] := z ;
END ListInitialize ;

PROCEDURE DeleteNext(t : INTEGER) =
BEGIN
    next[t] := next[next[t]] ;
END DeleteNext ;
```

```
PROCEDURE InsertAfter(v : ItemType ; t : INTEGER) =
BEGIN
    x := x + 1 ;
    key[x] := v ;
    next[x] := next[t] ;
    next[t] := x ;
END InsertAfter ;
```

Anmerkungen:

- `InsertAfter` testet nicht, ob das Array schon voll ist.
- Mit `DeleteNext` freigegebene Positionen im Array werden nicht wiederverwendet. (Dafür müßte zusätzlich eine Freispeicherliste verwaltet werden.)

Vor- und Nachteile der verschiedenen Implementierungen

sequentielles Array:

– vorgegebene maximale Größe

– sehr starr: Der Aufwand für Insert und Delete wächst linear mit der Größe der Liste.

– Daher nur bei „statischer“ Objektmenge zu empfehlen, d.h. wenn relativ wenige Insert- und Delete-Operationen nötig sind.

+ belegt keinen zusätzlichen Speicherplatz

Cursor-Darstellung:

– vorgegebene maximale Größe

– spezielle Freispeicherverwaltung erforderlich

+ Fehlerkontrolle leichter

Pointer-Implementierung:

+ maximale Anzahl der Elemente offen

+ Freispeicherverwaltung wird vom System übernommen

– Overhead durch Systemaufrufe

– Fehlerkontrolle schwierig

Einschub: Vergleich von Pseudocode und Java

Um die vorangegangenen Erläuterungen weiter zu vertiefen und einen Überblick über die Realisierung der dynamischen Datenstrukturen in Java zu geben, werden im folgenden die Implementierungen in Pseudocode und Java gegenübergestellt und miteinander verglichen.

Elementare (atomare) Datentypen. Die folgende Tabelle gibt einen Überblick über die atomaren Datentypen in Pseudocode und Java:

Pseudocode	Java	
INTEGER	int	Ganze Zahlen
CARDINAL	-	Natürliche Zahlen
CHARACTER	char	Zeichen (Buchstaben etc.)
REAL	double	Dezimalzahlen
BOOLEAN	boolean	Wahrheitswerte (TRUE oder FALSE)

Die Aufzählungstypen aus Pascal bzw. Modula-3 sind in Java nicht vorhanden.

Zusammengesetzte Typen. Als zusammengesetzte Datentypen gibt es in Java das Feld und den Verbund (Klasse). Im Vergleich zu Pseudocode werden diese folgendermaßen verwendet:

	Pseudocode	Java
Feld	`x: ARRAY[1..N] of INTEGER;`	`int [] x;` `x=new int[N]`

Verbund/Struktur	`TYPE complex = RECORD`
	`real : REAL;`
	`imag : REAL;`
	`END;`
		`class complex {`
		`double real;`
		`double imag;`
		`}`

Bemerkungen:

- In Pseudocode beginnen Felder oftmals mit dem Index 1, in Java jedoch immer mit dem Index 0.
- Die Klassen in Java bieten zusätzlich zu den Datenfeldern die Möglichkeit, Methoden an die Objekte eines Typs zu binden.

Zeiger. In Java werden Referenzierung und Dereferenzierung von Objekten automatisch durchgeführt. Java erlaubt keinerlei direkte Manipulation von Zeigern oder

Speicheradressen. Parameter an Funktionen werden in Java automatisch als Referenzen übergeben. Daher sind die per Referenz übergebenen Objekte manipulierbar, nicht jedoch die Referenzen selbst.

In Java gibt es wie in Modula-3 Garbage Collection. D.h. Speicher, der nicht mehr verwendet wird, wird automatisch freigegeben.

Gegenüberstellung einer Implementierung von Listen in Pseudocode und in Java.

Pseudocode	Wirkung	Java

`TYPE link = REF node ;`
`     node = RECORD`
`       key : ItemType;`
`       next : link;`
`     END;`
	key
	next
		`class link {`
		`  ItemType key;`
		`  link next;`
		`}`

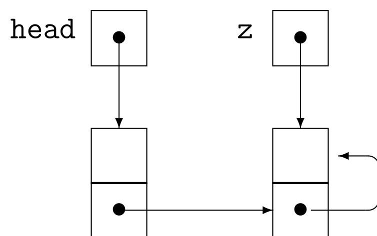
`VAR p, q : link ;`	p q	`link p,q;`
`p := NEW(link);`	p q	`p=new link();`
`p^.key:=x;`	p q x	`p.key=x;`
`q := NEW(link) ;`	p q x	`q=new link();`

Pseudocode	**Wirkung**	**Java**
`p^.next := q ;` `q^.next := NIL ;`	p q x	`p.next=q;` `q.next=null;`
`q^.key := z;`	p q x z	`q.key=z;`
`p^.next := NIL;` `q := NIL ;`	p q x	`p.next=null;` `q=null;`

Erstellen einer leeren Liste:

Pseudocode	**Java**

`PROCEDURE ListInitialize() =`
`BEGIN`
`    head := NEW(link) ;`
`    z    := NEW(link) ;`
`    head^.next := z ;`
`    z^.next    := z ;`
`END ListInitialize ;`
	`void ListInitialize()`
	`{`
	` head=new link();`
	` z=new link();`
	` head.next=z;`
	` z.next=z;`
	`}`



Nachfolger von `t^` löschen:

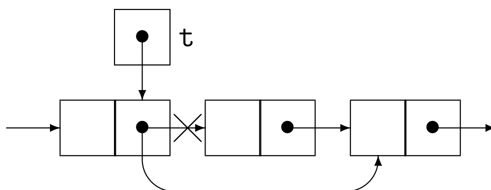
<table>
<tr><th>Pseudocode</th><th>Java</th></tr>
<tr><td>

```
PROCEDURE DeleteNext(t : link) =
BEGIN
    t^.next := t^.next^.next ;
END DeleteNext ;
```

</td><td>

```
void DeleteNext(link t)
{
  t.next=t.next.next;
}
```

</td></tr>
</table>



Nachfolger von `t^` einfügen:

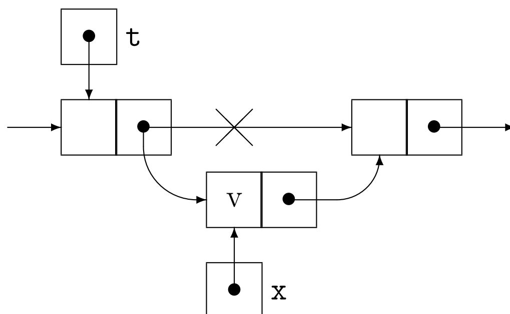
<table>
<tr><th>Pseudocode</th><th>Java</th></tr>
<tr><td>

```
PROCEDURE InsertAfter(v : ItemType ;
                      t : link) =
VAR x : link ;
BEGIN
    x := NEW(link) ;
    x^.key := v ;
    x^.next := t^.next ;
    t^.next := x ;
END InsertAfter ;
```

</td><td>

```
void InsertAfter(ItemType v,
                 link t)
{
  link x;
  x=new link();
  x.key=v;
  x.next=t.next;
  t.next=x;
}
```

</td></tr>
</table>



1.3.3 Stacks

Einen Stack kann man sich als Stapel von Elementen vorstellen (siehe Abb. 1.4). Stacks speichern (so wie Listen) Elemente in sequentieller Ordnung, der Zugriff ist jedoch auf das erste Element der Folge beschränkt. Die zulässigen Operationen sind:

Push : Neues Element wird am oberen Ende (Top) hinzugefügt.

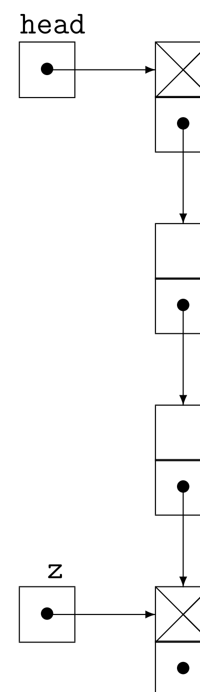
Pop : Oberstes Element wird vom Stack entfernt und zurückgeliefert.

Da das von Pop zurückgelieferte Element immer dasjenige ist, das *zuletzt* eingefügt wurde, spricht man auch von einem LIFO-Memory (last-in, first-out). Andere Bezeichnungen für „Stack“ sind „Stapel“, „Kellerspeicher“ und „pushdown stack“.

Implementierung mit Pointern

Der Stack verwendet die gleiche Datenstruktur wie die Liste, nur die Operationen unterscheiden sich folgendermaßen:

```
PROCEDURE Push(v : ItemType) =
VAR t : link ;
BEGIN
    t := NEW(link) ;
    t^.key := v ;
    t^.next := head^.next ;
    head^.next := t ;
END Push ;

PROCEDURE Pop() : ItemType =
VAR t : link ;
BEGIN
    t := head^.next ;
    head^.next := t^.next ;
    RETURN t^.key ;
END Pop ;

PROCEDURE StackEmpty() : BOOLEAN =
BEGIN
    RETURN head^.next = z ;
END StackEmpty ;
```



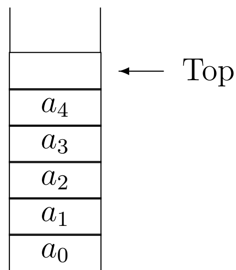


Abbildung 1.4: Schematische Darstellung eines Stacks

Implementierung mittels Array:

```
CONST stack_max = 100 ;
VAR   stack : ARRAY [0..stack_max] OF ItemType ;
      top : CARDINAL ;

PROCEDURE StackInitialize() =
BEGIN top := 0 ;
END StackInit ;

PROCEDURE Push(v : ItemType) =
BEGIN
    stack[top] := v ;
    top := top + 1 ;
END Push ;

PROCEDURE Pop() : ItemType =
BEGIN
    top := top - 1 ;
    RETURN stack[top] ;
END Pop ;

PROCEDURE StackEmpty() : BOOLEAN =
BEGIN
    RETURN top = 0 ;
END StackEmpty ;

PROCEDURE StackFull() : BOOLEAN =
BEGIN
    RETURN top > stack_max ;
END StackFull ;
```

Beachte: Über- und Unterlauf des Stacks müssen abgefangen werden.

1.3.4 Queues

Geläufige Namen für „Queue“ sind auch „Warteschlange“ und „Ringspeicher“. Bei der Queue ist das Einfügen und Entfernen von Elementen nur wie folgt möglich:

Put : Neues Element wird am Ende der Queue hinzugefügt.

Get : Vorderstes Element wird aus der Queue entfernt und zurückgeliefert.

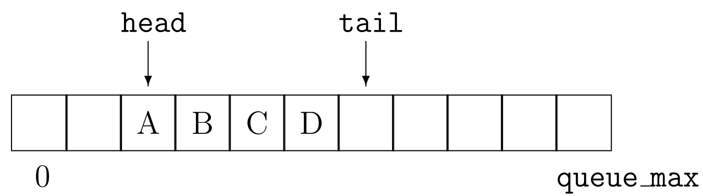
Da das von Get zurückgelieferte Element immer dasjenige ist, das *zuerst* eingefügt wurde, spricht man auch von einem FIFO-Memory (first-in, first-out).

Implementierung mittels Array:

```
CONST queue_max = 100 ;
VAR   queue : ARRAY [0..queue_max] OF ItemType ;
      head, tail : CARDINAL ;

PROCEDURE QueueInitialize() =
BEGIN
    head := 0 ;
    tail := 0 ;
END QueueInitialize ;

PROCEDURE Put(v : ItemType) =
BEGIN
    queue[tail] := v ;
    tail := tail + 1 ;
    IF tail > queue_max THEN tail := 0 ; END ;
END Put ;

PROCEDURE Get() : ItemType =
VAR t : ItemType ;
BEGIN
    t := queue[head] ;
    head := head + 1 ;
    IF head > queue_max THEN head := 0 ; END ;
    RETURN t ;
END Get ;

PROCEDURE QueueEmpty() : BOOLEAN =
BEGIN
    RETURN head = tail ;
END QueueEmpty ;
```

```
PROCEDURE QueueFull() : BOOLEAN =
BEGIN
    RETURN head = (tail+1) MOD (queue_max+1) ;
END QueueFull ;
```

Beachte: Über- und Unterlauf der Queue müssen abgefangen werden.



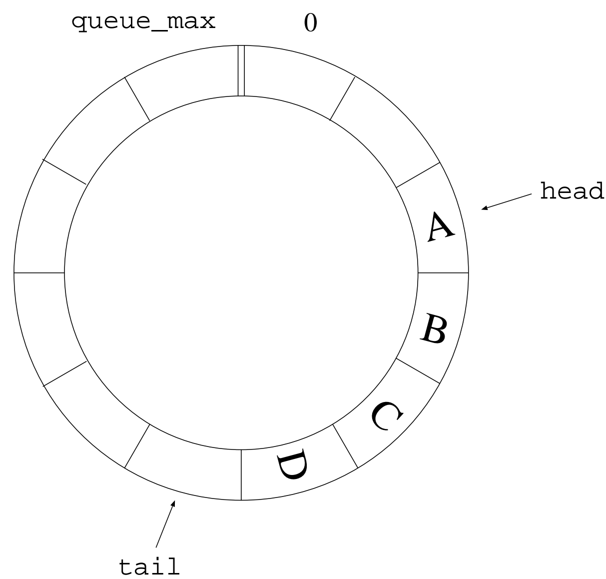


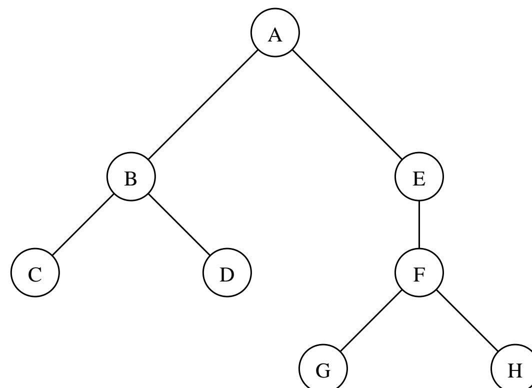
Abbildung 1.5: Schematische Darstellung eines Ringpuffers (Queue)

1.3.5 Bäume

Ein *Baum* besteht aus einer Menge von Knoten und einer Relation, die über der Knotenmenge eine Hierarchie definiert:

- Jeder Knoten, außer dem Wurzelknoten (*Wurzel*, root), hat einen unmittelbaren Vorgänger (*Vater*, parent, father).
- Eine *Kante* (Zweig, Ast, branch) drückt die Beziehung unmittelbarer Nachfolger und Vorgänger aus.
- Der *Grad* eines Knotens ist die Zahl seiner unmittelbaren Nachfolger.
- Ein *Blatt* (Blattknoten, leaf) ist ein Knoten ohne Nachfolger, also mit Grad 0. Mit anderen Worten: außer den Blattknoten hat jeder Knoten mindestens einen unmittelbaren Nachfolger.
- *Siblings* (*Brüder*, Geschwister) sind alle unmittelbaren Nachfolger des gleichen Vaterknotens.
- Ein *Pfad* (Weg) ist eine Folge von Knoten $n_1, \ldots, n_k$, wobei n_i unmittelbarer Nachfolger von n_{i-1} ist. (Es gibt keine Zyklen.)
  Die *Länge eines Pfades* ist die Zahl seiner Kanten = die Zahl seiner Knoten -1.
- Die *Tiefe* oder auch *Höhe eines Knotens* ist die Länge des Pfades von der Wurzel zu diesem Knoten.
- Die *Höhe eines Baumes* ist die Länge des Pfades von der Wurzel zum tiefsten Blatt.
- Der *Grad eines Baumes* ist das Maximum der Grade seiner Knoten.
  Spezialfall mit Grad = 2: Binärbaum.

Beispiel

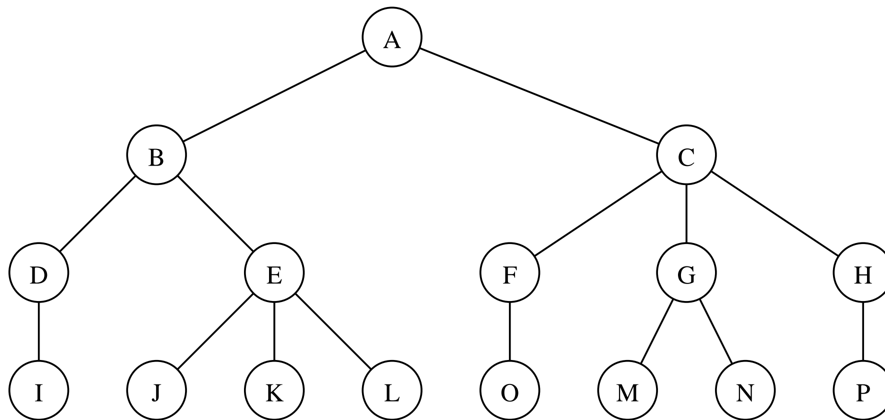


Wurzel	:	A
Vater	:	A ist Vater von E und B
Brüder	:	B und E sind Brüder
Blätter	:	C, D, G, H
Tiefe	:	F hat Tiefe 2
Höhe des Baumes	:	3
Pfad von A nach G	:	(A, E, F, G)

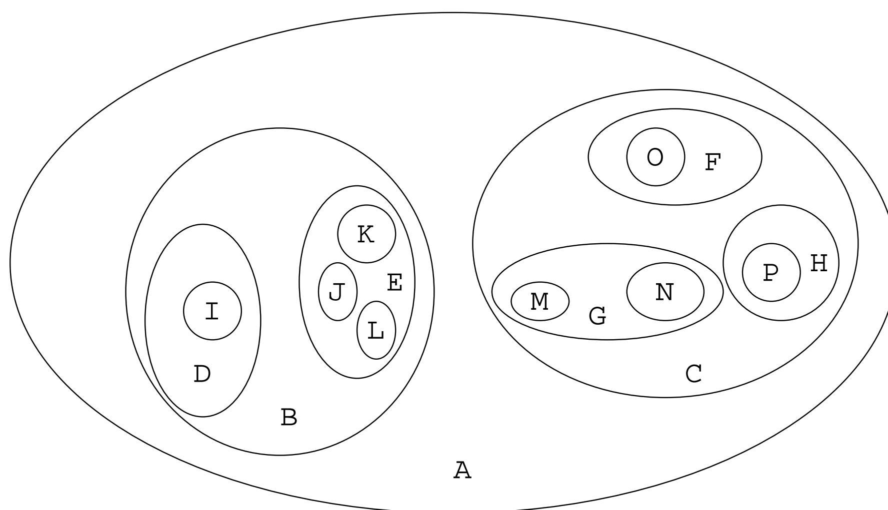
Graphische Darstellungen

Bäume können auf unterschiedliche Arten dargestellt werden. Die vier folgenden Darstellungen beschreiben dieselbe hierarchische Struktur:

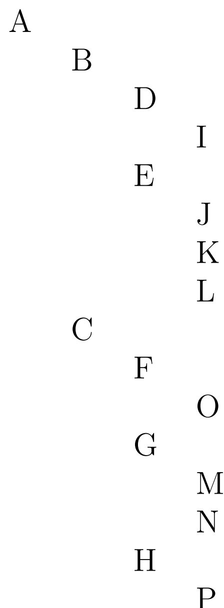
1. Darstellung als Graph



2. geschachtelte Mengen



3. Darstellung durch Einrückung



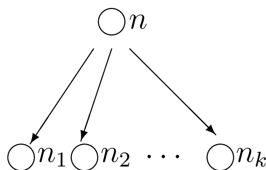
4. geschachtelte Klammern

(A(B(D(I),E(J,K,L)),C(F(O),G(M,N),H(P))))

Induktive Definition

1. Ein einzelner Knoten ist ein Baum. Dieser Knoten ist gleichzeitig der Wurzelknoten.

2. Sei n ein Knoten und seien $T_1, T_2, ..., T_k$ Bäume mit den zugehörigen Wurzelknoten $n_1, n_2, ..., n_k$. Dann wird ein neuer Baum konstruiert, indem $n_1, n_2, ..., n_k$ zu unmittelbaren Nachfolgern von n gemacht werden. T_k heißt dann k-ter Teilbaum (Unterbaum) des Knotens n.



Minimale und maximale Höhe eines Baumes

Sei T ein Baum vom Grad $d \geq 2$ und n Knoten, dann gilt:

- maximale Höhe $= n - 1$
- minimale Höhe $= \lceil \log_d (n(d-1)+1) \rceil - 1 \leq \lceil \log_d n \rceil$,

Beweis

maximale Höhe: klar, da es in T nur n Knoten gibt, es also maximal $n-1$ Kanten geben kann. Ein Baum mit maximaler Höhe ist zu einer linearen Liste entartet.

minimale Höhe: Ein *maximaler Baum* ist ein Baum, der bei gegebener Höhe h und gegebenem Grad d die maximale Knotenzahl $N(h)$ hat.

Also gilt für die maximale Knotenzahl $N(h)$:

$$\begin{aligned} N(h) &= 1 + d + d^2 + ... + d^h \quad \text{(geometrische Reihe)} \\ &= \frac{d^{h+1}-1}{d-1} \end{aligned}$$

Bei fester Knotenzahl hat ein maximal gefüllter Baum minimale Höhe. Ein Baum, dessen Ebenen bis auf die letzte komplett gefüllt sind, hat offensichtlich minimale Höhe. Also liegt die Anzahl n der Knoten, die man in einem Baum gegebener minimaler Höhe h unterbringen kann, zwischen der für einen maximalen Baum der Höhe h und einem maximalen Baum der Höhe $h-1$.

$$\begin{array}{rcccl} N(h-1) & < & n & \leq & N(h) \\ \frac{d^h-1}{d-1} & < & n & \leq & \frac{d^{h+1}-1}{d-1} \\ d^h & < & n(d-1)+1 & \leq & d^{h+1} \\ h & < & \log_d (n(d-1)+1) & \leq & h+1 \end{array}$$

und wegen $n(d-1)+1 < nd \;\; \forall n > 1$ folgt:

$$\begin{aligned} h &= \lceil \log_d (n(d-1)+1) \rceil - 1 \\ &\leq \lceil \log_d (nd) \rceil - 1 \\ &= \lceil \log_d n \rceil \end{aligned}$$

Spezialfall $d = 2$ (Binärbaum):

$$h = \lceil \mathrm{ld}(n+1) \rceil - 1 \leq \lceil \mathrm{ld}\, n \rceil$$

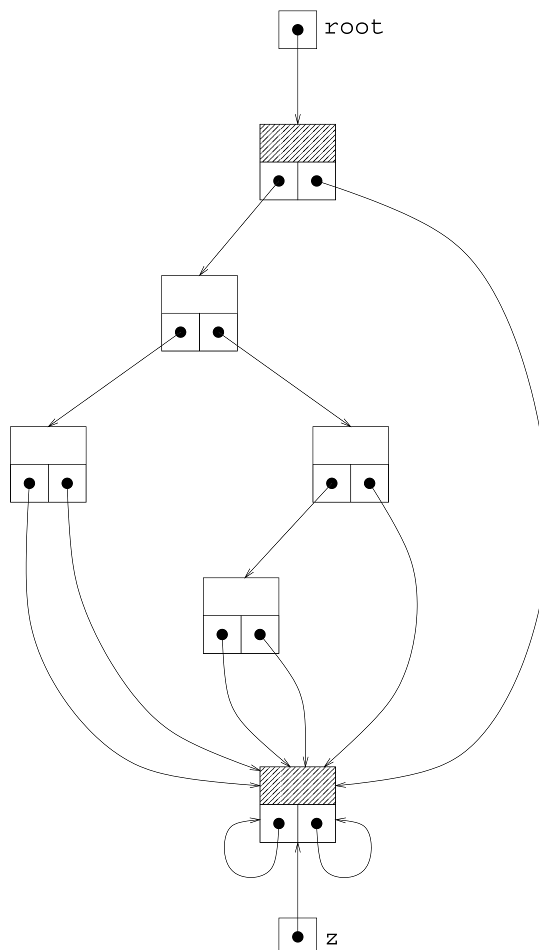
Implementierung von Binärbäumen

Zwei geläufige Implementierungen von Binärbäumen sind die Pointer-Darstellung und die Array-Einbettung.

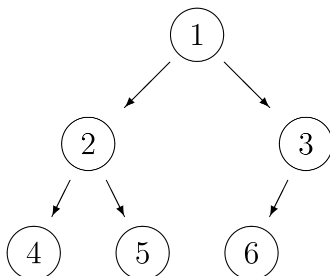
Pointer-Realisierung

```
TYPE node = REF RECORD
                  key : ItemType ;
                  left, right : node ;
            END ;
```

Zur Kennzeichnung der Wurzel und der Blätter werden (vgl. Implementierung der verketteten Listen) zwei spezielle Knoten `root` und `z` verwendet.



Array-Einbettung
Die Array-Einbettung eigent sich am besten zur Darstellung vollständiger Bäume. Ein Binärbaum heißt *vollständig*, wenn alle Ebenen bis auf die letzte vollständig besetzt sind und die letzte Ebene von links nach rechts aufgefüllt ist (auch: links-vollständig).



Numeriert man die Knoten wie hier gezeigt ebenenweise durch, so lassen sich die Positionen von Vorgängern und Nachfolgern auf folgende Weise einfach berechnen:

Vorgänger: $\text{Pred}(n) = \left\lfloor \frac{n}{2} \right\rfloor$

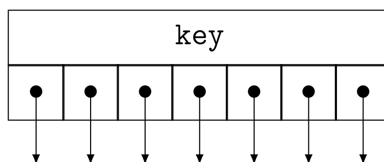
Nachfolger: $\text{Succ}(n) = \begin{cases} 2n & \text{linker Sohn} \\ 2n+1 & \text{rechter Sohn} \end{cases}$ (falls diese existieren)

Man prüft die Existenz der Nachfolger über die Arraygrenzen.

Implementierung von Bäumen mit Grad $d > 2$

Array von Pointern
Darstellung durch Knoten, die ein Array von Pointern auf ihre Nachfolgeknoten enthalten.

```
TYPE node = REF RECORD
                key  : ItemType ;
                sons : ARRAY [1..d] OF node ;
            END ;
```



Nachteile

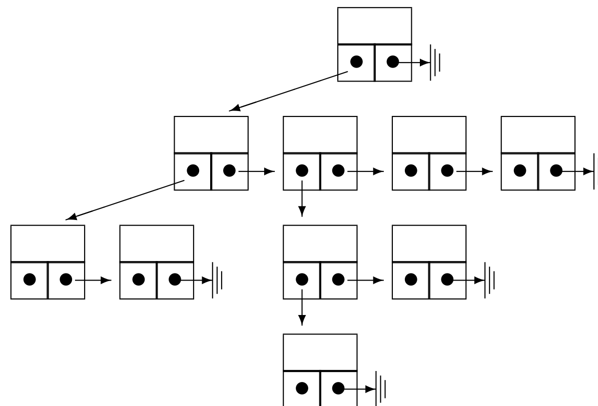
- der maximale Grad ist vorgegeben
- Platzverschwendung, falls der Grad der einzelnen Knoten stark variiert.

Darstellung durch Pseudo-Binärbaum

Strategie: Geschwister sind untereinander zu einer linearen Liste verkettet. Jeder Knoten verweist nur auf sein erstes Kind und seinen rechten Bruder (leftmost-child, right-sibling).

```
TYPE node = REF RECORD
                key : ItemType ;
                leftmostchild : node ;
                rightsibling  : node ;
            END ;
```

`leftmostchild` zeigt also auf das erste Element einer verketteten Liste der Geschwister mit den Nachfolgezeigern `rightsibling`.



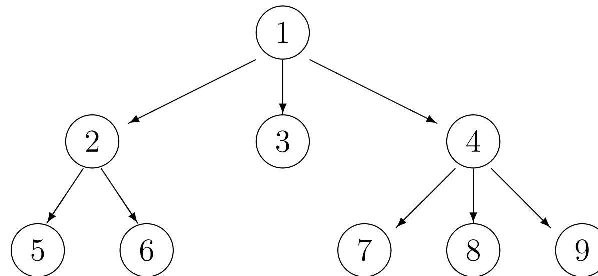
Array-Einbettung

Die Knoten werden ebenenweise von links nach rechts durchnumeriert (*breadth first*) und in Reihenfolge der Numerierung in einem Array abgelegt. Geschwisterknoten stehen dann an aufeinanderfolgenden Positionen. Zum Auffinden der Vorgänger und Nachfolger dienen folgende Arrays:

```
VAR Parent,
    LeftMostChild,
    RightMostChild : ARRAY [1..N] OF CARDINAL ;
```

Der Index 0 wird für nicht vorhandene Knoten verwendet.

Beispiel



i	`Parent`	`LeftMostChild`	`RightMostChild`
1	0	2	4
2	1	5	6
3	1	0	0
4	1	7	9
5	2	0	0
6	2	0	0
7	4	0	0
8	4	0	0
9	4	0	0

Durchlauf-Funktionen sind in dieser Darstellung sehr einfach:

- Durchlaufen aller Knoten: $i = 1, \ldots, n$
- Durchlaufen der Nachfolger eines Knotens i:
  $k =$ `LeftMostChild[`i`],..., RightMostChild[`i`]`

Durchlaufen eines Baums

Für viele Anwendungen ist ein kompletter Durchlauf aller Knoten eines Baumes notwendig (Durchmusterung, Traversierung, tree traversal).

Tiefendurchlauf (*depth first*): Zu einem Knoten n werden rekursiv seine Teilbäume $n_1, \ldots, n_k$ durchlaufen. Je nach Reihenfolge, in der der Knoten und seine Teilbäume betrachtet werden, ergeben sich:

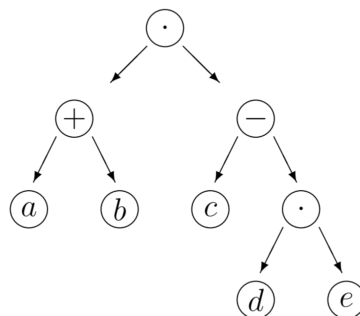
1. Preorder (Prefixordnung, Hauptreihenfolge):
   - betrachte n
   - durchlaufe $n_1 ... n_k$
2. Postorder (Postfixordnung, Nebenreihenfolge):
   - durchlaufe $n_1 ... n_k$
   - betrachte n
3. Inorder (Infixordnung, symmetrische Reihenfolge):
   - durchlaufe n_1
   - betrachte n
   - durchlaufe $n_2 ... n_k$

   Diese Ordnung ist nur für Binärbäume gebräuchlich.

Breitendurchlauf (*breadth first*, *level order*): Knoten werden ihrer Tiefe nach gelistet, und zwar bei gleicher Tiefe von links nach rechts.

Beispiel: Operatorbaum

Bei Anwendung auf einen Operatorbaum realisieren die unterschiedlichen Traversierungsarten die Darstellung des entsprechenden algebraischen Ausdrucks in verschiedenen Notationen. Beispiel: Der Ausdruck $(a + b) \cdot (c - d \cdot e)$ wird durch folgenden Operatorbaum repräsentiert:



Ausgabe aller Knoten mittels

- Preorder-Traversierung: $\cdot + ab - c \cdot de$
  (Prefixnotation, „polnische Notation“)
- Postorder-Traversierung: $ab + cde \cdot - \cdot$
  (Postfixnotation, „umgekehrte polnische Notation“)
- Inorder-Traversierung: $a + b \cdot c - d \cdot e$.
  (Infixnotation)

Rekursiver Durchlauf für Binärbäume

Wir wollen die genannten Tiefendurchläufe für Binärbäume in Pointer-Darstellung implementieren.

Preorder-Durchlauf

```
PROCEDURE Traverse(t : node) =
BEGIN
    IF t <> z THEN
        visit(t) ;
        traverse(t^.left) ;
        traverse(t^.right) ;
    END ;
END Traverse ;
```

Postorder-Durchlauf

```
PROCEDURE Traverse(t : node) =
BEGIN
    IF t <> z THEN
        traverse(t^.left) ;
        traverse(t^.right) ;
        visit(t) ;
    END ;
END Traverse ;
```

Inorder-Durchlauf

```
PROCEDURE Traverse(t : node) =
BEGIN
    IF t <> z THEN
        traverse(t^.left) ;
        visit(t) ;
        traverse(t^.right) ;
    END ;
END Traverse ;
```

Nicht-rekursiver Durchlauf für Binärbäume

Tiefendurchlauf in Preorder Reihenfolge (Preorder-Durchlauf, benötigt einen Stack)

```
PROCEDURE Traverse(t : node) =
VAR n : node ;
BEGIN
    Push(t) ;
    REPEAT
        n := Pop() ;
        IF n <> z THEN
            visit(n) ;
            Push(n^.right) ;
            Push(n^.left) ;
        END ;
    UNTIL StackEmpty() ;
END Traverse ;
```

Man beachte, daß zuerst der *rechte* Teilbaum auf den Stack geschoben wird.

Breitendurchlauf (Level-Order-Durchlauf, benötigt eine Queue)

```
PROCEDURE Traverse(t : node) =
VAR n : node ;
BEGIN
    Put(t) ;
    REPEAT
        n := Get() ;
        IF n <> z THEN
            visit(n) ;
            Put(n^.left) ;
            Put(n^.right) ;
        END ;
    UNTIL QueueEmpty() ;
END Traverse ;
```

1.3.6 Abstrakte Datentypen (ADT)

Wenn man eine Datenstruktur nur durch die auf ihr zugelassenen Operationen definiert, und die spezielle Implementierung ignoriert, dann gelangt man zu einem sogenannten *abstrakten* Datentyp. Ein abstrakter Datentyp (ADT) wird definiert durch:

1. eine Menge von Objekten (Elementen)
2. Operationen auf diesen Elementen (legen die Syntax eines Datentyps fest)
3. Axiome (Regeln) (definieren die Semantik des Datentyps)

Die Abstraktion von der konkreten Implementierung erleichtert die Analyse von Algorithmen erheblich.

Die Idee des abstrakten Datentyps wird in Modula-3 durch die Trennung in sichtbare Schnittstellen (`INTERFACE`) und verborgene Implementierung (`MODULE`) unterstützt. Dies erhöht auch die Übersichtlichkeit von umfangreichen Programmpaketen.

Weiterführende Literatur: [Güting, Seiten 21–23], [Sedgewick 88, Seite 31]

Beispiel: Algebraische Spezifikation des Stacks

- Sorten (Datentypen)
  Stack (hier zu definieren)
  Element („ElementTyp“)

- Operationen

StackInit:		$\rightarrow$	Stack	(Konstante)
StackEmpty:	Stack	$\rightarrow$	Boolean	
Push:	Element $\times$ Stack	$\rightarrow$	Stack	
Pop:	Stack	$\rightarrow$	Element $\times$ Stack	

- Axiome
  x: Element („ElementTyp“)
  s: Stack

$$\begin{aligned}
\text{Pop}(\text{Push}(x, s)) &= (x, s) \\
\text{Push}(\text{Pop}(s)) &= s \qquad \text{für StackEmpty}(s) = \text{FALSE} \\
\text{StackEmpty}(\text{StackInit}) &= \text{TRUE} \\
\text{StackEmpty}(\text{Push}(x, s)) &= \text{FALSE}
\end{aligned}$$

Realisierung [Sedgewick 88, Seite 27].

Anmerkung: Situationen, in denen undefinierte Operationen, wie z.B. Pop(StackInit), aufgerufen werden, erfordern eine gesonderte Fehlerbehandlung.

Potentielle Probleme von ADTs

- Bei komplexen Fällen wird die Anzahl der Axiome sehr groß.
- Die Spezifikation ist nicht immer einfach zu verstehen.
- Es ist schwierig zu prüfen, ob die Gesetze vollständig und widerspruchsfrei sind.

Wir werden im weiteren aus folgenden Gründen nicht näher auf die abstrakte Spezifikation von Datentypen eingehen:

- Wir betrachten hier nur relativ kleine und kompakte Programme.
- Uns interessieren gerade die verschiedenen möglichen Implementierungen, die durch die abstrakte Spezifikation verborgen werden sollen.
- Wir wollen die Effizienz der konkreten Implementierung untersuchen, wobei ADTs gerade nicht helfen.