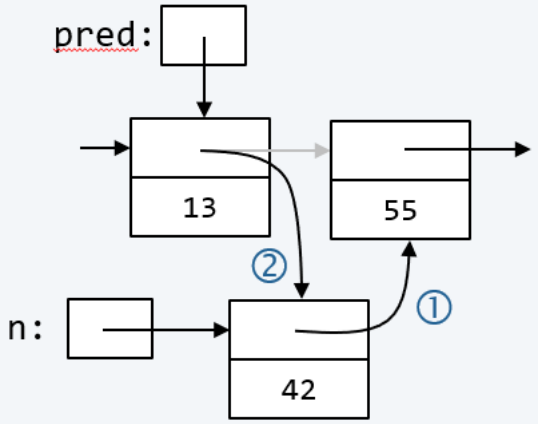
6 Dynamische Datenstrukturen



6.1 Grundlegendes

- Bisher haben wir Datenobjekte verwendet, deren Struktur und Speicherbedarf zum Zeitpunkt der Deklaration festgelegt wurden
- Struktur und Größe einer Datenstruktur können während der Ausführung eines Algorithmus nicht mehr geändert werden
- Beispiel

```
type
  Entry = compound
    word: string
    pages: array [1:10] of int
  end
  Index = array [1:100] of Entry
```

Feldgröße kann zur Laufzeit nicht verändert werden, nur Füllstand.

- Die richtige Feldgröße lässt sich im Voraus oft nicht abschätzen
- Konsequenz: Der Speicherbereich ist entweder zu klein (Einschränkung) oder zu groß (Verschwendung)

Dynamische Datenobjekte: Ziel und Begriff

Ziel

- Bereitstellung von Konzepten und Konstrukten, mit denen Datenobjekte gebildet werden können, deren Struktur und Größe während Ausführung von Algorithmen/Programmen veränderbar ist

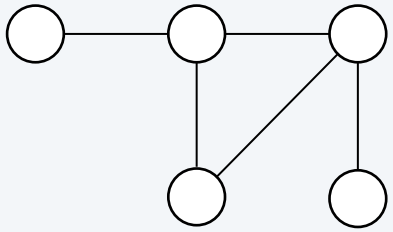
Begriff: Unter dynamischen Datenobjekten verstehen wir Datenobjekte, die dadurch charakterisiert sind,

- dass sie aus Komponenten bestehen, die miteinander in Beziehung stehen, also vernetzt sind und
- für die gilt, dass zur Laufzeit eines Algorithmus, sowohl die Anzahl und Werte der einzelnen Komponenten als auch die Beziehungsstruktur zwischen den Komponenten geändert werden können

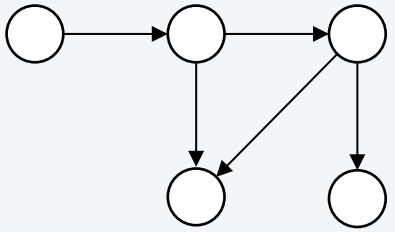
Dynamische Datenobjekte können als Einheit betrachtet werden, die in ihrer „Größe“ wachsen und schrumpfen kann

Dynamische Datenobjekte: Aufbau und Bestandteile

- Dynamische Datenobjekte sind gerichtete Graphen



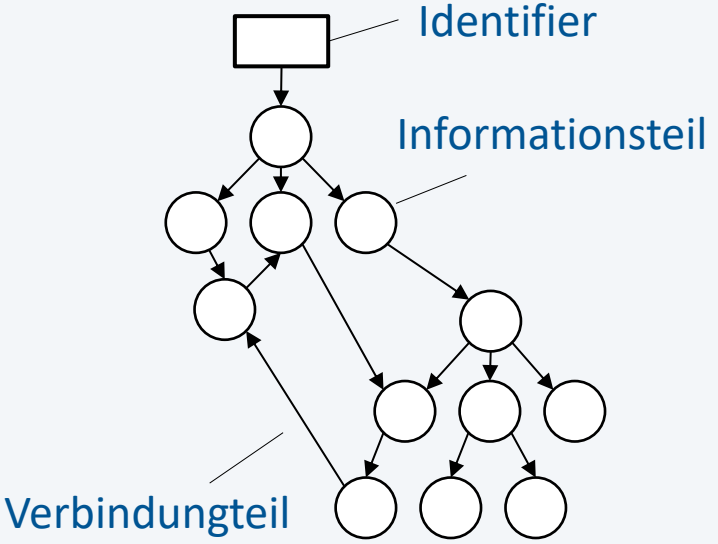
Graph



Gerichteter Graph

- Bestehen aus einem Bezeichner (Identifier) zur Identifikation und zum Zugriff auf das Datenobjekt als Ganzes,
- aus Informationsteilen (Knoten, compound) zur Aufnahme der Daten und
- aus Verbindungsteilen (Zeiger) zur Modellierung des Beziehungsgeflechts

Neuer Datentyp!

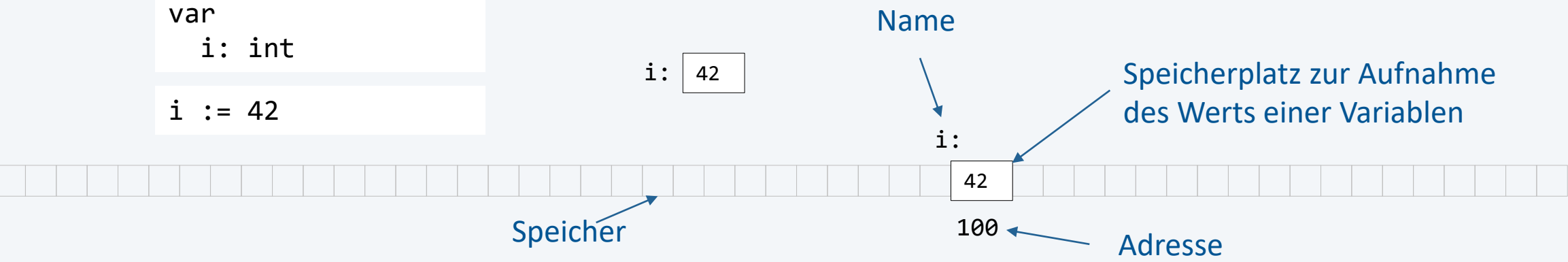


6.2 Das Konzept des Zeigertyps (pointer type)

Variablen enthalten Werte (z.B. `42`, `27.15`, `"Hallo"`, `'X'`, `true`)

```
var
  i: int
```

```
i := 42
```



- Jeder lokalen Variablen ist bestimmter Speicherplatz zugeordnet, der zum Zeitpunkt der Algorithmenaktivierung angelegt und durch eine Adresse, die mit dem Namen der Variablen assoziiert ist, identifiziert wird
- Adressen entsprechen positiven, ganzzahligen Werten, die man in Variablen ablegen kann
- Zeigervariablen (kurz Zeiger) enthalten Adressen von anderen Variablen, oder allgemein von Datenobjekten
- Zeiger sind also Datenobjekte, deren Datentyp ein Zeigertyp ist

eigener Typ, eigener Wertebereich

Zeigertyp (*pointer type*)

Zeigertyp ist ein Datentyp, dessen Wertebereich die Adressen eines Speichers (Speicherbereichs) umfasst

Wertebereich ist also abhängig vom Hauptspeicher: z.B. 4gb ram hat 2^32 bit, also braucht man 4 byte

Zur Deklaration von Variablen mit einem Zeigertyp führen wir das Konstrukt → ein:

```
type
  TPtr = →T
var
  p: TPtr
```

oder

```
var
  p: →T
```

- Die Variable p ist ein Zeigerobjekt, das auf ein T-Datenobjekt verweist, ihr Wert ist also eine Adresse
- Werte vom Typ TPtr sind Adressen von Datenobjekten des Typs T

gelten als unterschiedliche Typen!

Beispiele

Bsp.: pi - enthält eine Adress von einer Variable vom Typ integer

```
var
  pi: →int
  pr: →real
  pc: →char
```

```
var
  ps: →string
  pa: →array [1:10] of int
```

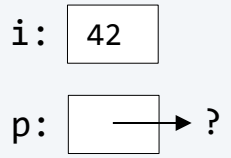
Die Datentypen von pi, pr, pc etc. sind nicht kompatibel!
Zum Beispiel ist
pi := pr
nicht möglich!

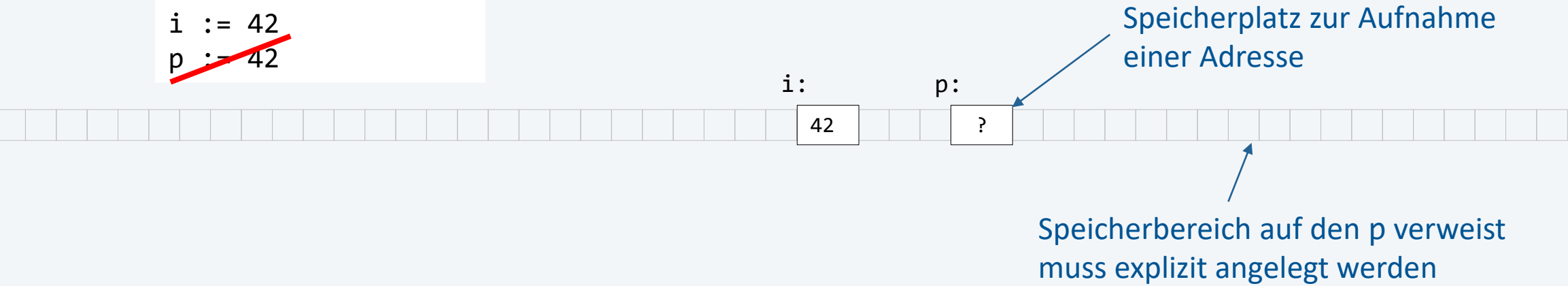
Zeigertyp (*pointer type*)

Der Speicher für Datenobjekte, auf die eine Zeigervariable zeigt, wird nicht bei der Deklaration, sondern erst zur Laufzeit festgelegt

```
var
  i: int
  p: →int
```

```
i := 42
p := 42
```





Datenobjekte, auf die über Zeigervariable zugegriffen wird, müssen explizit „erzeugt“ werden und können explizit vom Speicher wieder entfernt werden (so wird das Wachsen und das Schrumpfen von Datenstrukturen möglich

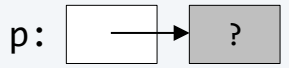
Erzeugen von über Zeiger referenzierten Datenobjekten

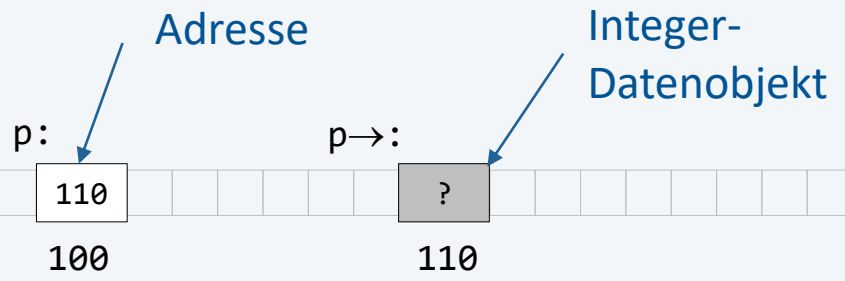
Zur Erzeugung von über Zeiger referenzierten Datenobjekten verwenden wir die Operation New

Variable, welche Adressen speichert, wird erzeugt. Sie bekommt also einen Platz im Speicher
New(int) gibt die Speicheradresse des referenzierten Datenobjekts zurück, welcher dann der Zeigervariable zugeordnet werden kann

```
var
  p: →int
```

```
p := New(↓int)
```



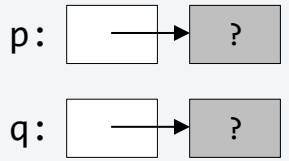


Die Standardoperation New allokiert Speicherplatz und gibt dessen Adresse (110) zurück

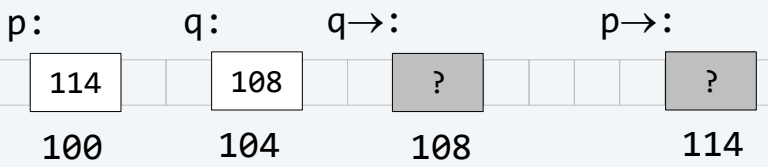
Beispiel:

```
var
  p, q: →int
```

```
p := New(↓int)
q := New(↓int)
```



Die Datenobjekte p→ und q→ existieren bis sie explizit wieder entfernt werden

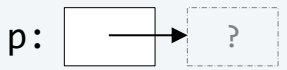


Entfernen von über Zeiger referenzierten Datenobjekten

Über Zeiger referenzierte Datenobjekte können explizit wieder entfernt werden, dazu benutzen wir die Operation `Dispose`

```
var
  p: →int
```

```
p := New(↓int)
…
Dispose(↓p)
```



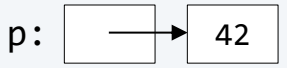
- Standardoperation `Dispose(↓p)` gibt den Speicherplatz frei, der über `p` referenziert wird; Adresse in `p` ist danach nicht mehr „gültig“; d.h. auf die Daten des Objektes, das über `p` referenziert wird, kann nicht mehr zugegriffen werden (das Objekt existiert nicht mehr)
- `Dispose` kann für jedes dynamische Datenobjekt nur einmal aufgerufen werden (für jedes `New` höchstens ein `Dispose`)
- Nach `Dispose` kann wiederum Speicherplatz mit `New` reserviert werden

Zugriff auf über Zeiger referenzierte Datenobjekte

Für den Zugriff auf über Zeiger referenzierte Datenobjekte führen wir den Operator → ein

```
var
  p: →int
```

```
p := New(↓int)
p→ := 42
```



Zeiger (Wert = Adresse 110) auf Speicherplatz vom Typ `int`

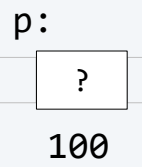
Speicherplatz (vom Typ `int`) beginnend bei Adresse 110

Zugriff auf Datenkomponenten setzt voraus, dass zuvor mit New ein Datenobjekt erzeugt (d.h. Speicherbereich dafür allokiert) wurde

```
var
  p: →int
```

```
p→ := 42  !!!
```

Zugriff mit Operator → über „ungültige“ Zeiger führt zu undefiniertem Verhalten, kann (leider) nicht vom Compiler geprüft werden



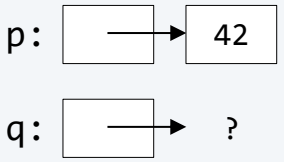
Erzeugung – Wertzuweisung – Entfernung

Dynamisch erzeugte Datenobjekte können von mehreren (beliebig vielen) Zeigervariablen (Bezeichnern) aus referenziert werden

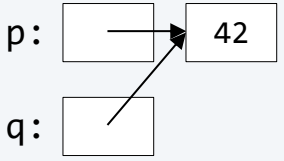
Speicherplatz darf genau einmal freigegeben werden

```
var
  p, q: →int
```

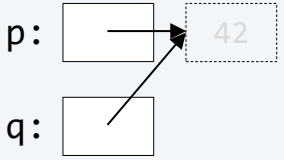
① `p := New(↓int)`
`p→ := 42`



② `q := p`



③ `Dispose(↓p)`



④ `Dispose(↓q)` !!!

Verwendung von `Dispose` mit "ungültigem" Zeiger führt zu undefiniertem Verhalten!

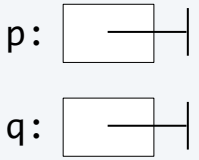
Spezieller Adresswert `null`

Null References: The Billion Dollar Mistake

Um auszudrücken, dass eine Zeigervariable noch nicht (oder nicht mehr) auf ein bestimmtes Datenobjekt zeigt, führen wir den speziellen (Adress-) Wert `null` ein

Der Zeigerwert `null` (graphisch durch das Symbol ⊣ dargestellt) dient zur Initialisierung von Zeigervariablen und zur korrekten Einstellung einer Zeigervariablen nach dem Entfernen eines assoziierten Datenobjekts

```
p := null
q := null
```



Beispiel: Typische Verwendungen

```
var
  p: →int
```

```
p := null
…
p := New(↓int)
…
```

```
if p ≠ null then
  … p→ …
end
```

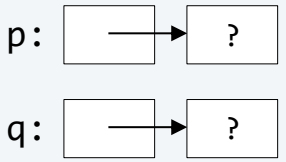
Nicht mehr erreichbare Datenobjekte

Speicherleichen (garbage)

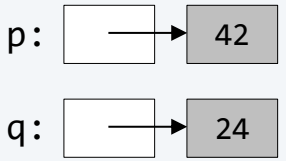
Mit New erzeugte (also dynamische) Datenobjekte können nur über Zeiger erreicht werden

Wenn man den Zeiger auf ein Datenobjekt „verliert“, kann dieses Objekt nicht mehr erreicht werden: Es existiert zwar noch, bildet aber eine sogenannte Speicherleiche (*garbage*)

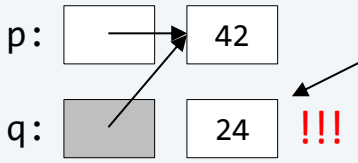
①
```
var
  p, q: →int

p := New(↓int)
q := New(↓int)
```



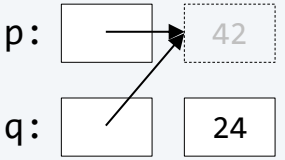
②
```
p→ := 42
q→ := 24
```



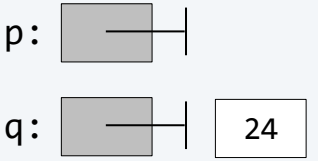
③
```
q := p
```



④
```
Dispose(↓p)
```



⑤
```
p := null
q := null
```



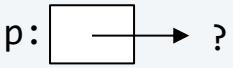
Beispiel für ein dynamisch erzeugtes Datenobjekt

Typdeklaration

```
type
  Person = compound
    surname, firstName: string
    …
  end -- Person
```

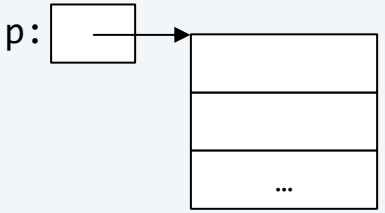
Variablendeklaration

```
var
  p: →Person
```



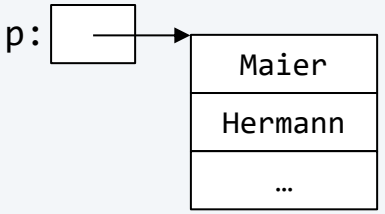
Objekt erzeugen

```
p := New(↓Person)
```



Zugriff auf Objektkomponenten

```
p→surname := "Maier"
p→firstName := "Hermann"
```



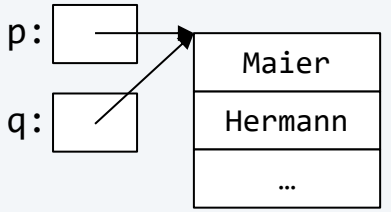
Beispiel für mehrfach referenzierte Datenobjekte

Beispiel

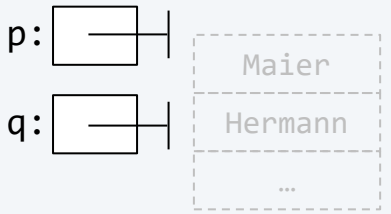
```
var
  p, q: →Person
```

```
p := New(↓Person)
p→surname := "Maier"
p→firstName := "Hermann"
q := p
Write(↓q→firstName)
```

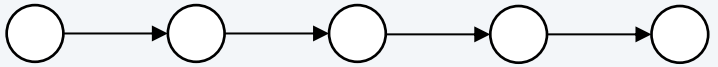
gibt ”Hermann” aus




```
Dispose(↓p)
p := null
q := null
```

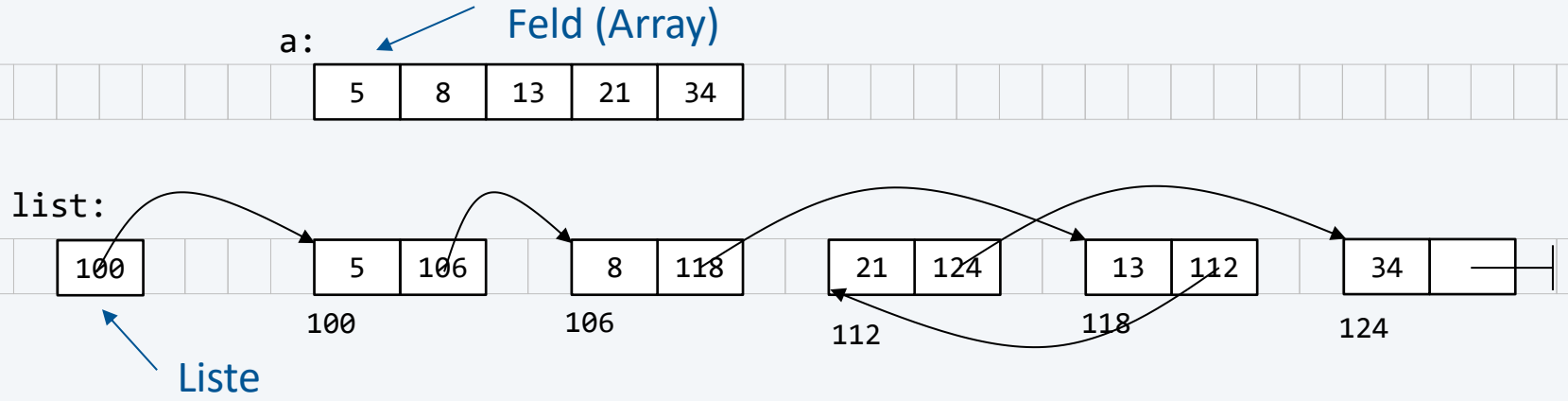


6.3 Einfach-verkettete Listen



Begriff

- Mittels verketteter Listen ist es möglich, eine sequenzielle Anordnung von sogenannten Knoten im Speicher zu bilden (vgl. eindimensionale Felder)
- Eine verkette Liste ist entweder leer oder repräsentiert eine sequenzielle Anordnung von Knoten gleichen Datentyps, deren Reihenfolge explizit durch Verkettung festgelegt ist
- Diese Verkettung wird durch in den Knoten enthaltene Zeiger realisiert, die bei einfach-verketteten Listen auf den jeweils nächsten Knoten, den Nachfolgerknoten verweisen



Verkettung von Datenobjekten

Jedes Datenobjekt (Listenknoten) enthält eine Zeiger- und eine (oder mehrere) Datenkomponente(n)

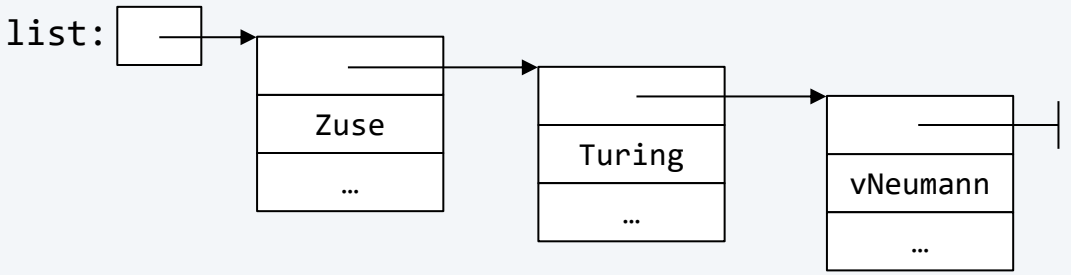
Jeder Listenknoten enthält also einen Zeiger (`next`) auf den nächsten Knoten (Nachfolgerknoten) zur Verkettung der Knoten zu einer Liste

Die gesamte Liste wird über eine Zeigervariable, zeigend auf den ersten Konten, referenziert (ihr Wert ist `null`, wenn die Liste leer ist)

Der letzte Listenknoten enthält in Komponente `next` den Wert `null`

Beispiel: Deklarationen für eine Liste mit Knoten für Personendaten

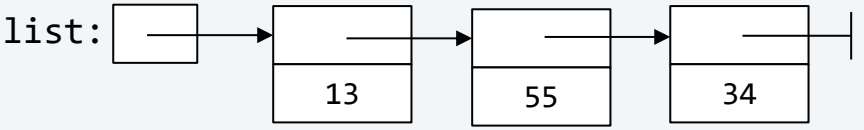
```
type
  Person = compound
    next: →Person
    name: string
    …
  end -- Person
var
  list: →Person
```



6.3.1 Einfach-verkettete Listen: Deklarationserfordernisse

Beispiele

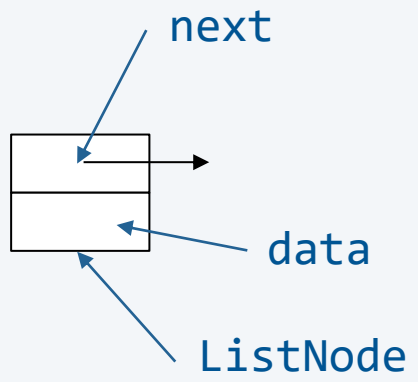
Leere Liste



Liste mit drei Knoten

Erforderliche Deklarationen für eine Liste (Beispiel Zahlenliste)

```
type
  ListNodePtr = →ListNode
  ListNode = compound
    next: ListNodePtr
    data: int
  end -- ListNode
  ListPtr = ListNodePtr
var
  list: ListPtr
```



Hinweis: Zur Verbesserung der Lesbarkeit führen wir den Typnamen `ListNodePtr` ein (Zeiger auf Knoten vom Datentyp `ListNode`)

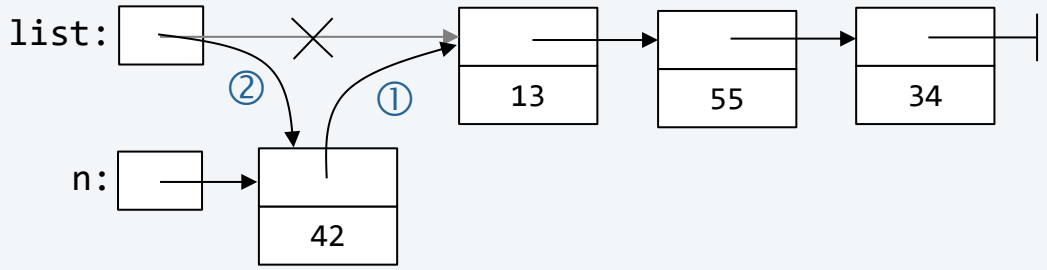
6.3.2 Einfach-verkettete Listen: Einfügen eines Knotens vorne

Ein neuer Knoten soll am Beginn der Liste (oft Listenkopf genannt) eingefügt werden, d.h. neuen Listenkopf bilden

```
var
  list: ListPtr
  n: ListNodePtr
```

```
n := New(↓ListNode)
n→data := 42
```

```
n→next := list    ①
list := n         ②
```



Anweisung `n→next := list` muss vor `list := n` erfolgen

Funktioniert auch für leere Liste (`list = null`)

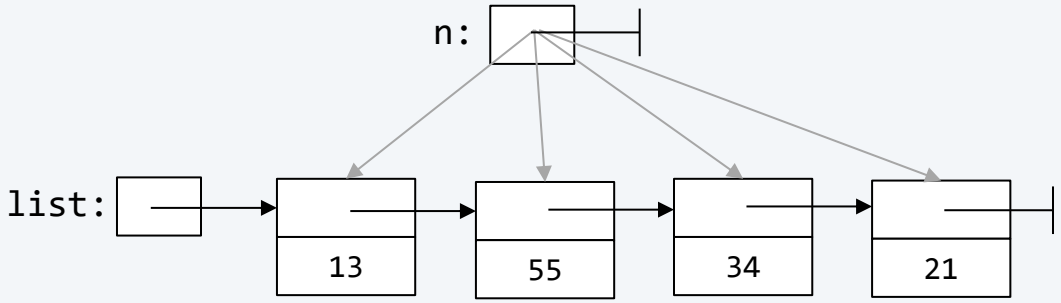
```
n→next := null
list := n
```

denn er neue Eintrag muss auch null drinnen haben - denn er ist gleichzeitig der letzte

6.3.3 Einfach-verkettete Listen: Durchwandern von Listen

Von vorne nach hinten

```
n := list
while n ≠ null do
  "bearbeite Knoten n"
  n := n→next
end
```



Knoten n_{i+1} (referenz. durch $n_i \rightarrow$next)

Beispiel: Ausgabe der Knotenwerte

```
WriteList(↓list: ListPtr)
  var n: ListNodePtr
begin
  n := list
  while n ≠ null do
    WriteLn(↓n→data)
    n := n→next
  end
end WriteList
```

Ausgabe:

```
13
55
34
21
```

Einfach-verkettete Listen – Durchwandern von Listen

Von hinten nach vorne

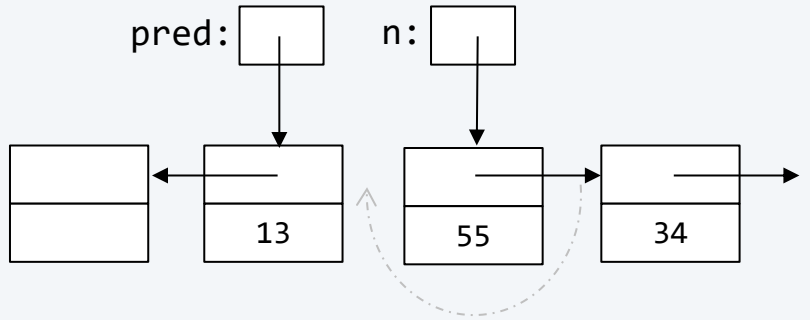
Trick bei einfacher Verkettung:

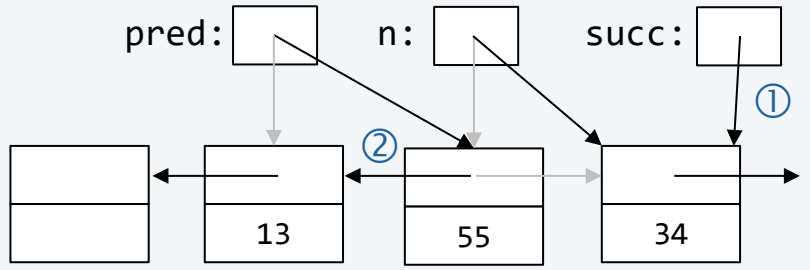
1. Umdrehen
2. Bearbeitung von vorne nach hinten
3. Umdrehen

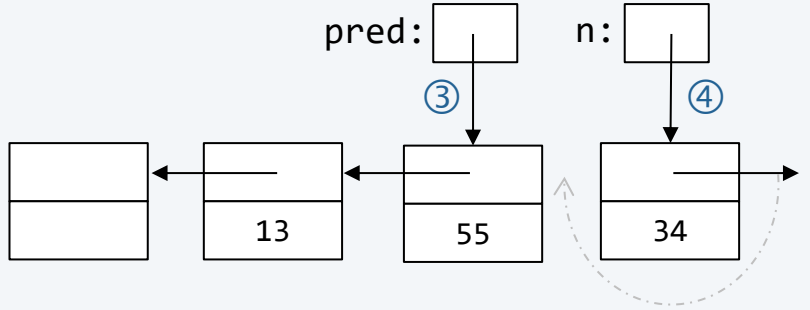
Umdrehen einer Liste

```
pred := null
n := list
while n ≠ null do
  succ := n→next      ①
  n→next := pred      ②
  pred := n           ③
  n := succ           ④
end -- while
list := pred
```

in Vorlesung übersprungen







Einfach-verkettete Listen: Suchen nach einem best. Knoten

Gesucht ist der Zeiger auf den Knoten, dessen Komponente `data` den Wert `x` enthält

```
FirstNodeWith(↓list: ListPtr ↓x: int): ListNodePtr
  var n: ListNodePtr
begin
  n := list
  while (n ≠ null) and (n→data ≠ x) do
    n := n→next
  end
  return n
end FirstNodeWith
```

gute Struktur
kann man auch lösen, in dem man aus einer Schleife mit return rausspringt, das ist aber unstrukturiert und kein guter Stil

hier wird dereferenziert - also es wird der wert angezeigt auf den der pointer n zeigt - das geht nur, wenn n ungleich null - also ist die Reihenfolge relevant, da von links nach rechts gelesen wird

Der Algorithmus arbeitet nur dann korrekt, wenn Ausdruck `(n ≠ null) and (n→data ≠ x)` von links nach rechts ausgewertet wird und bei `n = null` der zweite Term nicht mehr ausgewertet wird

Kurzschlussauswertung

- **`false and`** `term` liefert immer `false`, `term` wird nicht mehr geprüft
- **`true or`** `term` liefert immer `true`, term wird nicht mehr geprüft

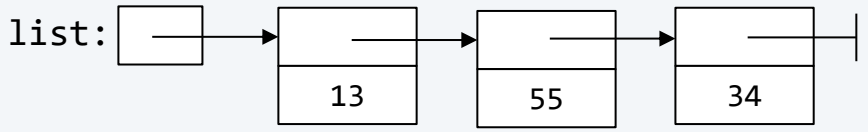
6.3.4 Einfach-verkettete Listen: Anfügen eines Knotens hinten

Ein neuer Knoten soll am Ende der Liste eingefügt werden

Neuen Knoten erzeugen

```
var
  list: ListPtr
  n: ListNodePtr
```

```
n := New(↓ListNode)
n→data := 42
n→next := null
```

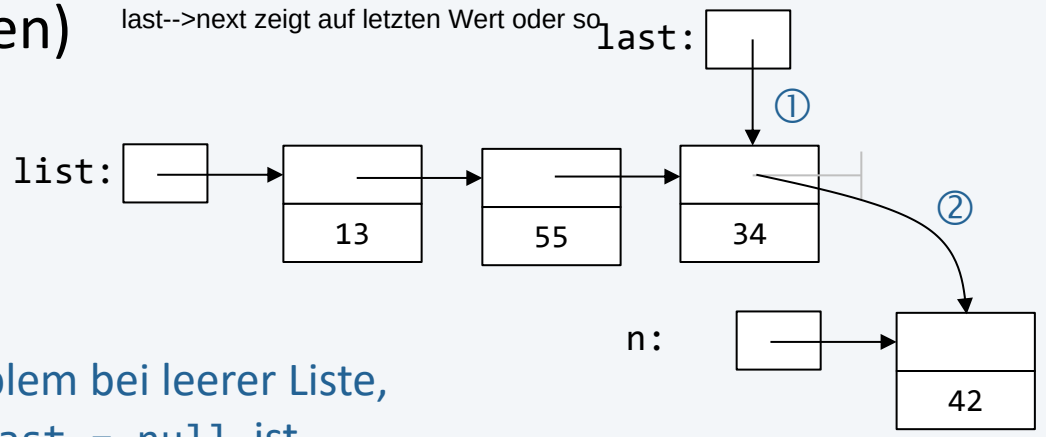


Suche das Listenende (letzter Knoten)

```
last := list
while last→next ≠ null do
  last := last→next
end
```

①

last-->next zeigt auf letzten Wert oder so



Neuen Knoten anfügen

```
last→next := n
```

②

Problem bei leerer Liste,
da `last = null` ist

Einfach-verkettete Listen: Anfügen eines Knotens hinten

Algorithmus zum Einfügen (Anfügen) eines Knotens am Ende der Liste

```
Append(↓↑list: ListPtr ↓n: ListNodePtr)
  var
    last: ListNodePtr
begin
  if list = null then
    list := n
  else
    last := list
    while last→next ≠ null do
      last := last→next
    end
    last→next := n
  end
end Append
```

Sonderfall für leere Liste

erlaubt, weil `list ≠ null` und damit auch `last ≠ null`

- Algorithmus ist ineffizient weil die gesamte Liste durchwandert werden muss
- Abhilfe durch zusätzlichen Zeiger `last`, der immer den letzten Knoten der Liste identifiziert ➔ hinten einfügen trivial

6.3.5 Einfach-verkettete Listen – Einfügen eines Knotens

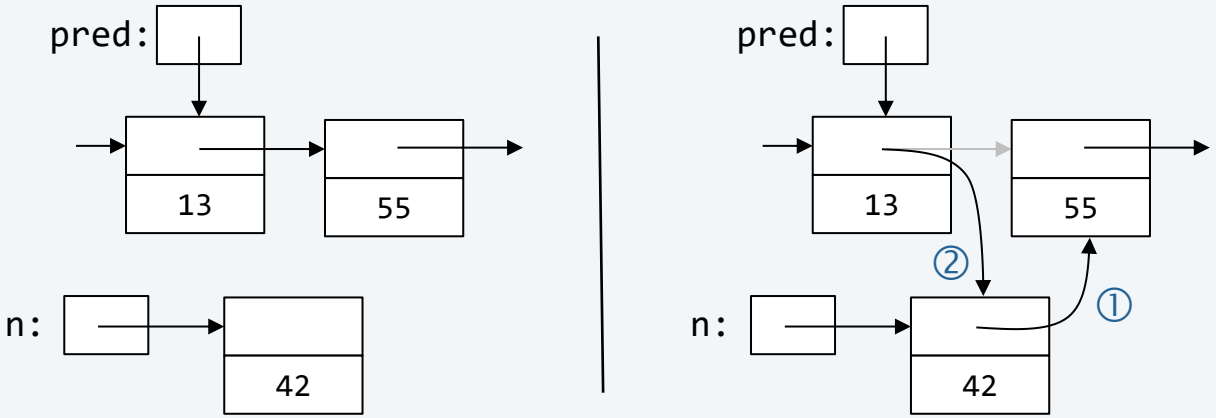
Ein neuer Knoten soll so in eine Liste eingefügt werden, dass eine bestimmte Ordnung erhalten bleibt (z.B. aufsteigend sortiert)

Der neue Knoten soll zwischen zwei Knoten eingefügt werden

```
n := New(↓ListNode)
n→data := 42
```

```
n→next := pred→next   ①
pred→next := n        ②
```

Reihenfolge auch hier relevant



Funktioniert, wenn Knoten für `pred` existiert (d.h. auch am Ende)

Wenn `pred = null` (leere Liste oder `n→data < list→data`)

```
n→next := list
list := n
```

Einfach verkettete Listen – Einfügen eines Knotens

Algorithmus zum Einfügen eines Knotens in eine sortierte Liste

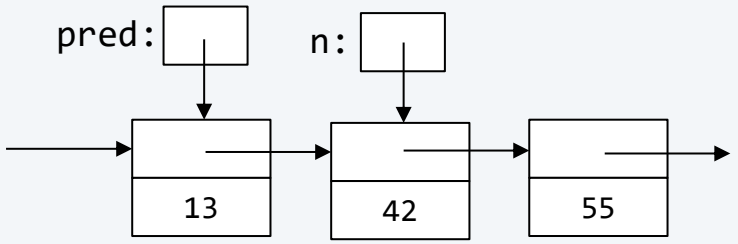
```
SortedInsert(↓↑list: ListPtr ↓n: ListNodePtr)
  var
    pred, succ: ListNodePtr
begin
  pred := null
  succ := list
  while (succ ≠ null) and (n→data > succ→data) do
    pred := succ
    succ := succ→next
  end
  if pred = null then
    list := n
  else -- pred ≠ null
    pred→next := n
  end
  n→next := succ
end SortedInsert
```

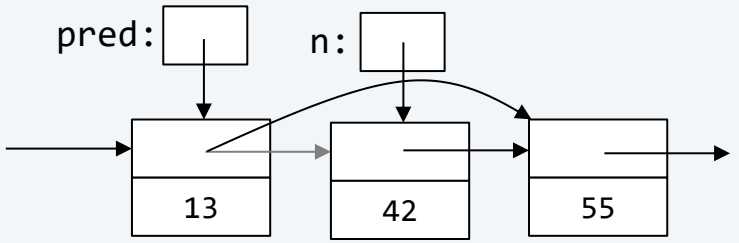
6.3.6 Einfach-verkettete Listen – Entfernen eines Knotens

Der erste Knoten, dessen Datenkomponente einen bestimmten Wert x enthält, soll aus der Liste `list` entfernt und gelöscht werden

Dazu muss Knoten n (mit Wert x) sowie dessen Vorgänger `pred` ermittelt werden

Entfernen des Knotens n





`pred→next := n→next`

Funktioniert, wenn Knoten für `pred` existiert

Wenn `pred = null` (n ist erster Knoten der Liste)

```
list := n→next
```

Einfach-verkettete Listen – Entfernen eines Knotens

Algorithmus

```
DeleteNodeWith(↓↑list: ListPtr ↓x: int)
  var pred, n: ListNodePtr
begin
  -- 1. search node n
  pred := null; n := list
  while (n ≠ null) and (n→data ≠ x) do
    pred := n
    n := n→next
  end
  -- 2. remove and dispose node
  if n ≠ null then
    if pred = null then
      list := n→next
    else
      pred→next := n→next
    end
    Dispose(↓n)
  end
end DeleteNodeWith
```

6.3.7 Eigenschaften einfach-verketteter Listen

- Können beliebig wachsen und schrumpfen
- Einfaches Durchwandern von vorne nach hinten
- Einfügen von Knoten einfach (kein Verschieben von Elementen der Datenstruktur erforderlich)
- Entfernen von Knoten einfach (kein Verschieben von Elementen der Datenstruktur erforderlich)
- Kein direkter Zugriff auf i-tes Element der Datenstruktur möglich

dadurch: binäre Suche nicht möglich

- Speicherbedarf je Element der Datenstruktur durch Komponente `next` erhöht

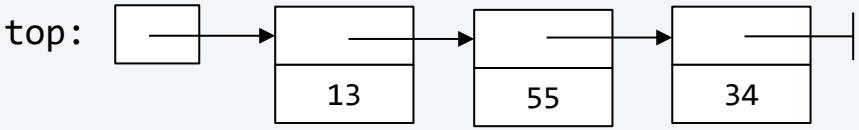
evtl. Klausuraufgabe: wie finde ich die mitte in einer Liste

6.4 Eine besondere Liste – der Kellerspeicher (Stack)

Ein Stack ist eine Datenstruktur, in die nur an einem Ende (*top*) Objekte eingefügt und wieder entfernt werden können, d.h. das zuletzt eingefügte Datenobjekt wird als erstes wieder entfernt

Einfache Realisierung durch einfach-verkettete Listen

Leerer Stack



Stack mit drei Elementen

Deklarationen

```
type
  StackNodePtr = →StackNode
  StackNode = compound
    next: StackNodePtr
    data: int
  end -- StackNode
  StackPtr = StackNodePtr

var
  top: StackPtr
```

Zeiger auf oberstes Element

Eine besondere Liste – der Kellerspeicher (Stack)

Operationen

- Die Operation `Push` „legt“ ein Datenobjekt auf den Stack
- Die Operation `Pop` „holt“ das oberste Datenobjekt vom Stack
- Mit der Operation `IsEmpty` kann geprüft werden, ob der Stack leer ist

Die Datenstruktur mit ihren Zugriffsalgorithmen lässt sich als Algorithmus mit Gedächtnis (`top` als Gedächtnisvariable) z.B. in Form eines Moduls realisieren

```
interface of IntStack
  Push(↓e: int)
  Pop(↑e: int)
  IsEmpty(): bool
end IntStack
```

```
implementation of IntStack
  var
    top: StackPtr
  …  -- siehe nächste Seiten
init
  top := null
end IntStack
```

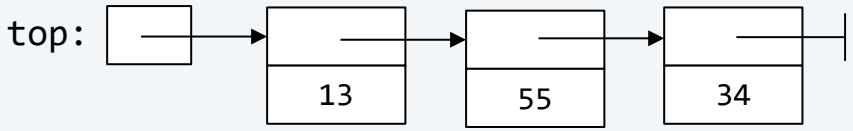
Eine besondere Liste – der Kellerspeicher (Stack)

Operation Push

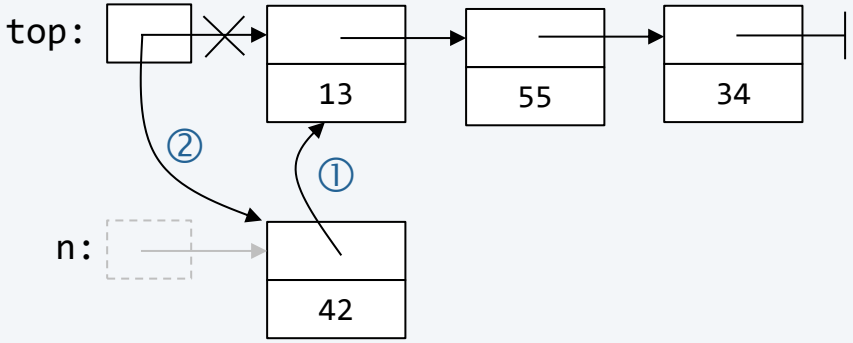
```
Push(↓e: int)
  var
    n: StackNodePtr
begin
  n := New(↓StackNode)
  n→data := e
  n→next := top   ①
  top := n        ②
end Push
```

Beispiel: Push(↓42)

Vor Aufruf:



Nach Aufruf:



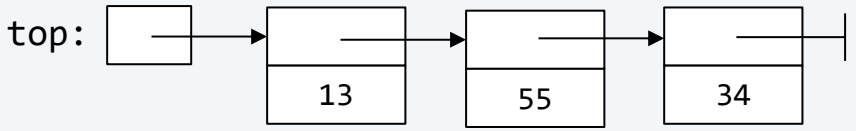
Eine besondere Liste – der Kellerspeicher (Stack)

Operation Pop

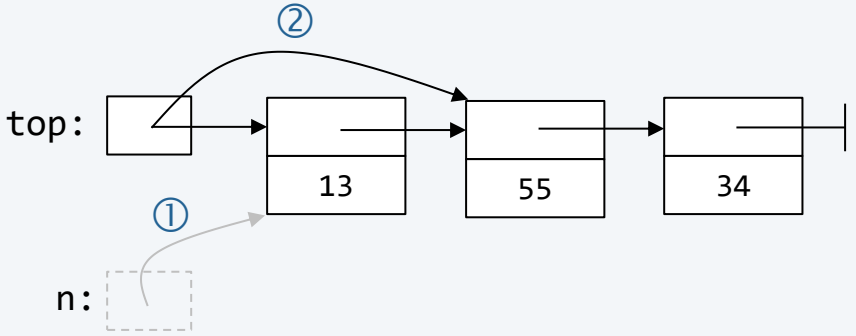
```
Pop(↑e: int)
  var
    n: StackNodePtr
begin
  if IsEmpty() then
    Write(↓"Fehler: ...")
    halt
  else
    n := top          ①
    e := n→data
    top := n→next     ②
    Dispose(↓n)
  end
end Pop
```

Beispiel: Pop(↑x)

Vor Aufruf:



Nach Aufruf:



Eine besondere Liste – der Kellerspeicher (Stack)

Operation `IsEmpty`

```
IsEmpty(): bool
begin
  return (top = null)
end Pop
```

Beispiel: Verwendung des Stacks

```
var
  e: int
```

```
Push(↓11)
Push(↓22)
Push(↓33)

while not IsEmpty() do
  Pop(↑e)
  WriteLn(↓e)
end
```

Ausgabe:

```
33
22
11
```

Eine besondere Liste – der Kellerspeicher (Stack)

Beispiel: Verwendung des Stacks

```
var
  n: int
```

```
Read(↑n)
while n > 0 do
  Push(↓(n mod 2))
  n := n div 2
end

while not IsEmpty() do
  Pop(↑e)
  Write(↓e)
end
```

n	Ausgabe
1	1
2	10
5	101
50	110010
255	11111111
256	100000000
5461	101010101010101

6.5 Anwendungsbeispiel für einfach-verkettete Listen

Es soll ein Modul (eine Datenkapsel) zur Verwaltung des Stichwort-verzeichnisses zu einem Dokument (z.B. zu einem Buch über Algorithmen und Datenstrukturen) entwickelt werden

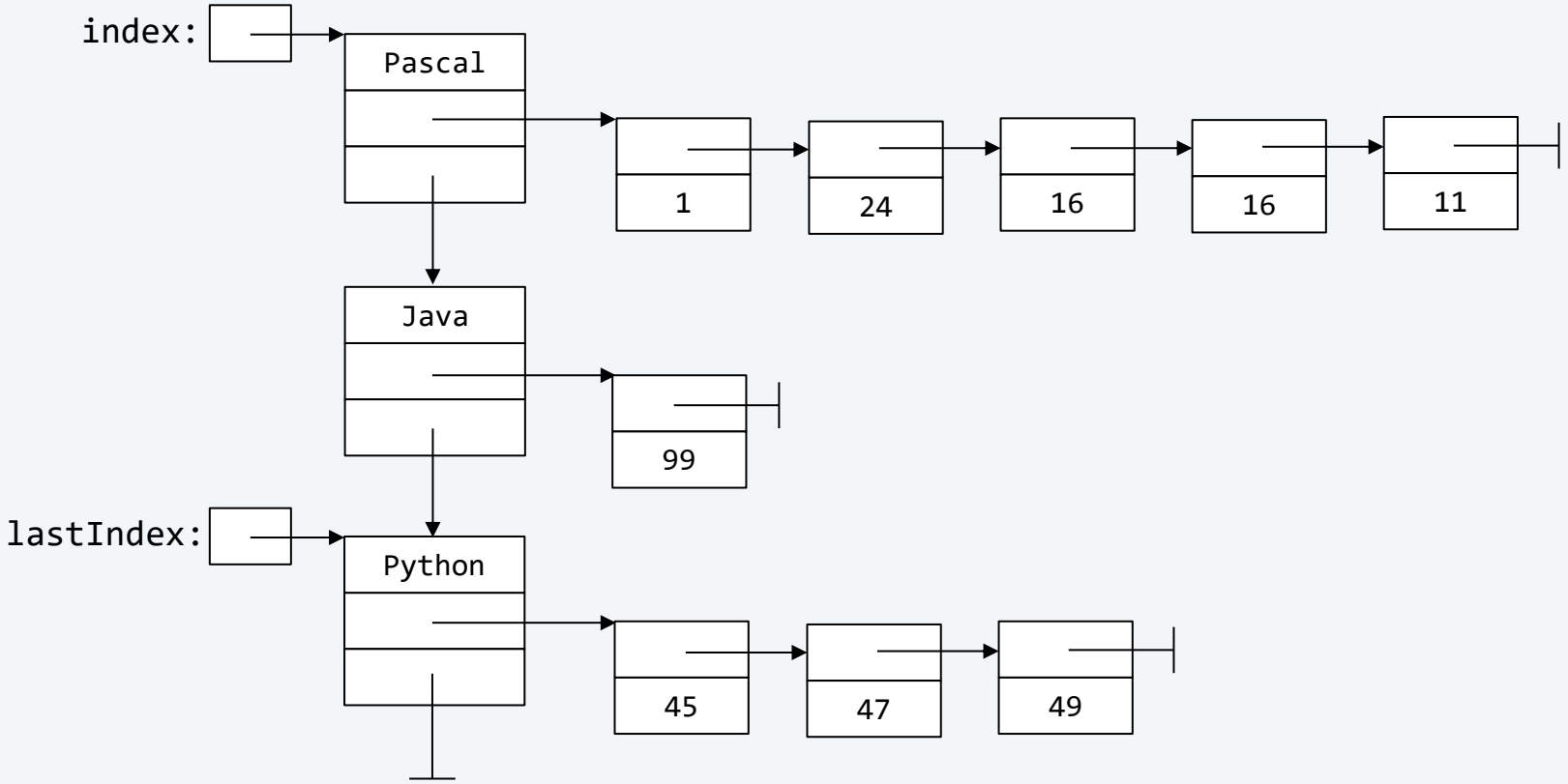
Der Einfachheit halber soll das Modul zunächst nur einen Algorithmus zum Aufbau eines Stichwortverzeichnisses, d.h. zum Einfügen eines Paares (Stichwort, Seitennummer) bereitstellen.

Um einfache Verhältnisse zu haben braucht das Stichwortverzeichnis – in Abweichung zur Realität – nicht sortiert zu sein.

```
interface of Index
  InsertKeyword(↓keyword: string ↓pageNr: int)
  SortIndex()
  PrintIndex()
  ...
end Index
```

Beispiel Stichwortverzeichnis – Lösungsidee Datenstruktur

Lösungsidee zur Wahl der Datenstruktur



- Sortierreihenfolge der Stichwörter und der Seitennummern beliebig
- Eine Seitennummer kann je Stichwort mehrfach vorkommen
- Jedes Stichwort kann nur einmal vorkommen

Beispiel Stichwortverzeichnis: erforderliche Deklarationen

Aus der Lösungsidee für die Datenstruktur zur Aufnahme des Stichwortverzeichnisses folgt, dass wir folgende Datentypen und Variablen benötigen:

```
type
  KeywordPtr = →KeywordNode
  KeywordNode = compound
    word: string
    pages: PagePtr
    next: KeywordPtr
  end -- KeywordNode
  PagePtr = →PageNode
  PageNode = compound
    next: PagePtr
    pageNr: int
  end -- PageNode
var
  index: KeywordPtr
  lastIndex: KeywordPtr
```

← Gedächtnisvariablen

Beispiel Stichwortverzeichnis – Entwurf `InsertKeyword`

Einfügen eines Paares (keyword, pageNr)

```
InsertKeyword(↓keyword: string ↓pageNr: int)
  var
    page, n: PagePtr
    entry: KeywordPtr
begin
  page := New(↓PageNode)
  page→next := null
  page→pageNr := pageNr
  FindKeywordNode(↓keyword ↑entry)
  if entry = null then
    InsertNewKeyword (↓keyword ↓page)
  else
    n := entry→pages
    while n→next ≠ null do
      n := n→next
    end
    n→next := page
  end
end InsertKeyword
```

neues Stichwort, noch nicht im Verzeichnis

Stichwort bereits im Verzeichnis;
Seitennummer hinzufügen

Beispiel Stichwortverzeichnis: Entwurf `InsertNewKeyword`

Ein neuer Knoten zur Aufnahme des noch nicht im Verzeichnis enthaltenen Stichwortes muss angelegt, gefüllt, mit dem Knoten für die Seitennummer verbunden und an das bestehende Stichwortverzeichnis (referenziert durch `index` und `lastIndex`) angefügt werden

```
InsertNewKeyword (↓keyword: string ↓page: PagePtr)
  var
    entry: KeywordPtr
begin
  entry := New(↓KeywordNode)
  entry→word := keyword
  entry→next := null
  entry→pages := page
  if lastIndex = null then
    index := entry
  else
    lastIndex→next := entry
  end
  lastIndex := entry
end InsertNewKeyword
```

Verzeichnis ist leer

neuer Eintrag ist erster Eintrag

Beispiel Stichwortverzeichnis – Entwurf `FindKeywordNode`

Suche eines Eintrags

```
FindKeywordNode(↓keyword: string ↑entry: KewordPtr)
begin
  entry := index
  while (entry ≠ null) and (entry→word ≠ keyword) do
    entry := entry→next
  end
end FindKeywordNode
```

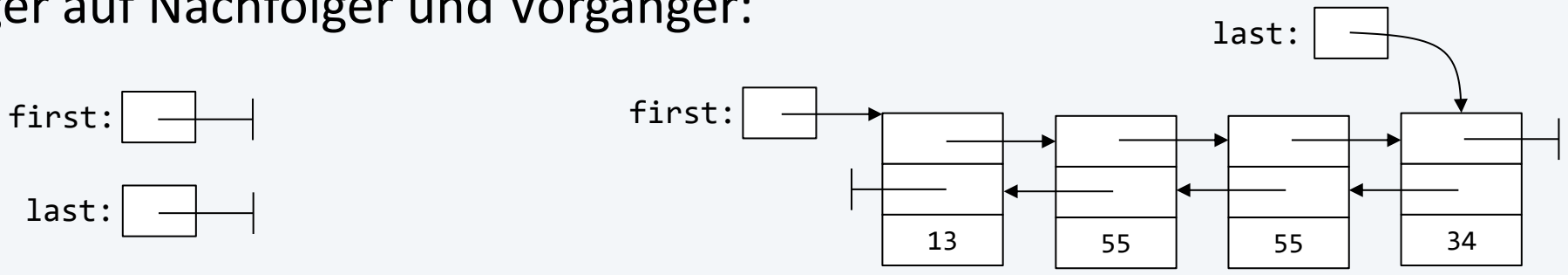
beginne Suche beim ersten Eintrag im Verzeichnis

6.6 Doppelt-verkettete Listen

Einfach-verkettete Listen haben den Nachteil, dass sie nur in einer Richtung durchwandert werden können, das wirkt sich ggf. negativ auf die Laufzeiteffizienz aus.

Doppelt-verkettete Listen können in beiden Richtungen durchwandert werden, das bietet mehr Flexibilität, wirkt sich aber negativ auf die Speichereffizienz aus.

Zeiger auf Nachfolger und Vorgänger:

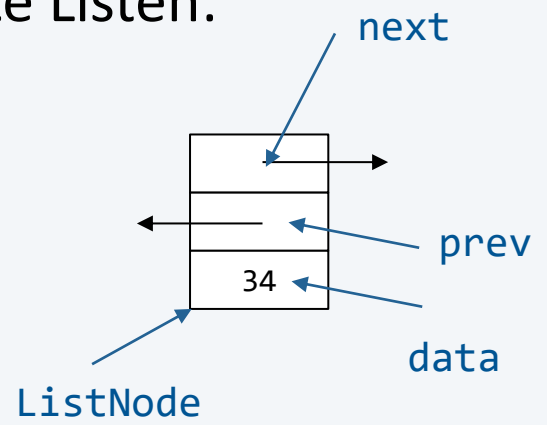


- Mit den Zeigervariablen `first` und `last` referenzieren wir Anfang und Ende der Liste (oft auch mit head und `tail` bezeichnet)
- Für leere Liste gilt `first = null` und `last = null`

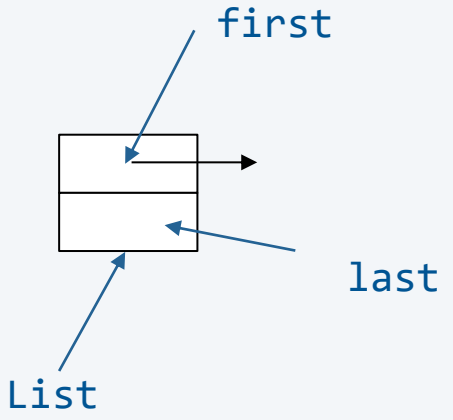
6.6.1 Doppelt-verkettete Listen: Deklarationserfordernisse

Erforderliche Deklarationen für doppelt-verkettete Listen:

```
type
  ListNodePtr = →ListNode
  ListNode = compound
    prev, next: ListNodePtr
    data: int
  end -- ListNode
```




```
type
  List = compound
    first, last: ListNodePtr
  end -- List
```

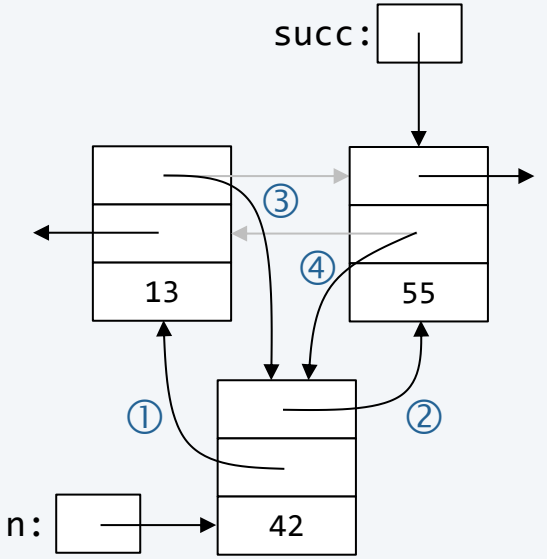


6.6.2 Doppelt-verkettete Listen – Einfügen eines Knotens

Ein neuer Knoten soll so in eine Liste eingefügt werden, dass eine bestimmte Ordnung erhalten bleibt (z.B. aufsteigend sortiert)

Der neue Knoten n soll zwischen zwei Knoten (vor succ) eingefügt werden

```
n→prev := succ→prev     ①
n→next := succ          ②
succ→prev→next := n     ③
succ→prev := n          ④
```



Funktioniert, wenn Knoten für succ und succ→prev existieren

Einfügen am Beginn (succ→prev = null) und Ende (succ = null) muss gesondert betrachtet werden (siehe Algorithmus)

Doppelt-verkettete Listen – Einfügen eines Knotens

Algorithmus zum Einfügen eines Knotens in eine sortierte Liste

Pfeil andersrum

```
SortedInsert(↓↑list: List ↑n: ListNodePtr)
  var
    succ: ListNodePtr
begin
  if list.first = null then
    list.first := n
    list.last := n
  else
    succ := list.first
    while (succ ≠ null) and (succ→data < n→data) do
      succ := succ→next
    end
    if succ = list.first then
       list.first→prev := n
       n→next := list.first
       list.first := n
    …
```

Nachfolger von n in sortierter Liste

Liste leer

Einfügestelle suchen

vorne einfügen

man geht davon aus, dass sowohl in first als auch in Last ein Null drinsteht

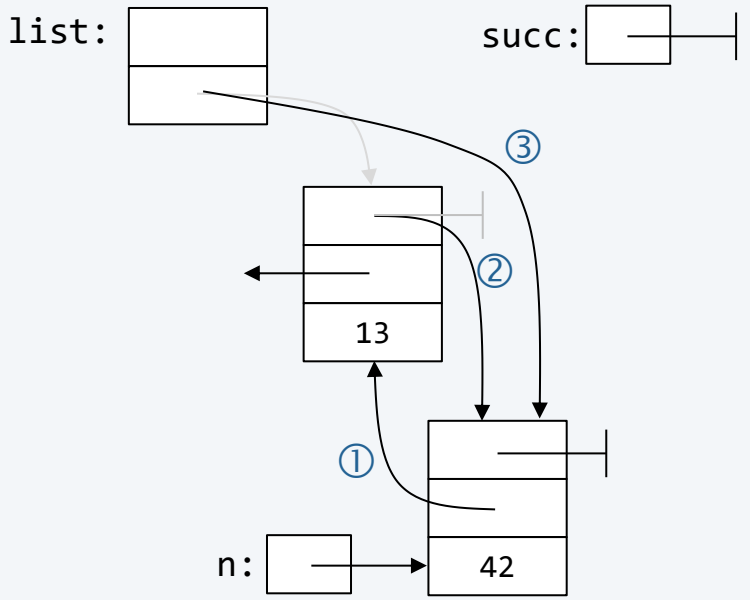
(hier nicht mehr leere Liste, mindestens 1 Knoten, weil vorher ausgeschlossen)

Doppelt-verkettete Listen – Einfügen eines Knotens

Algorithmus zum Einfügen eines Knotens in eine sortierte Liste

```
    …
    elsif succ = null then          ← hinten anfügen
      n→prev := list.last           ①
      list.last→next := n           ②
      list.last := n                ③
    else                            ← innerhalb der Liste einfügen
      n→prev := succ→prev
      n→next := succ
      succ→prev→next := n
      succ→prev := n
    end
  end
end SortedInsert
```

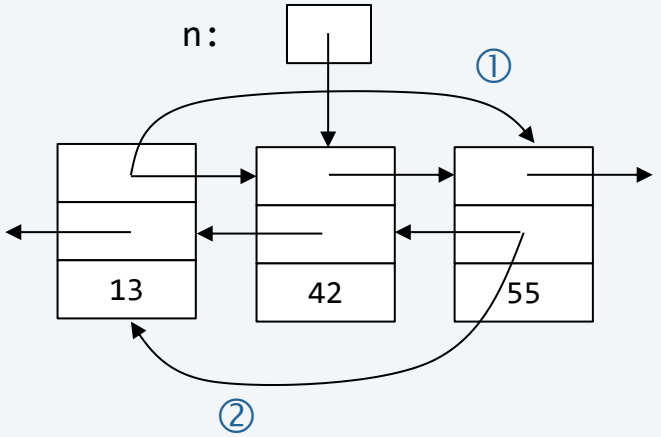
Hinten anfügen:



6.6.3 Doppelt-verkettete Listen: Entfernen eines Knotens

Der Knoten n soll aus der Liste entfernt werden

```
n→prev→next := n→next  ①
n→next→prev := n→prev  ②
n→prev := null
n→next := null
```



Funktioniert, wenn Vorgängerknoten (n→prev) und Nachfolgerknoten (n→next) existieren

Doppelt-verkettete Listen: Entfernen eines Knotens

Algorithmus zum Entfernen eines Knotens

```
Delete(↓↑list: List ↓n: ListNodePtr)
begin
  assert n ∈ list
  if (list.first = n) and (n = list.last) then
    list.first := null; list.last := null
  elsif list.first = n then
    list.first := list.first→next
    list.first→prev := null
  elsif list.last = n then
    list.last := list.last→prev
    list.last→next := null
  else
    n→prev→next := n→next
    n→next→prev := n→prev
  end
  n→prev := null
  n→next := null
  -- ggf. Dispose(↓n)
end Delete
```

entferne einzigen Knoten

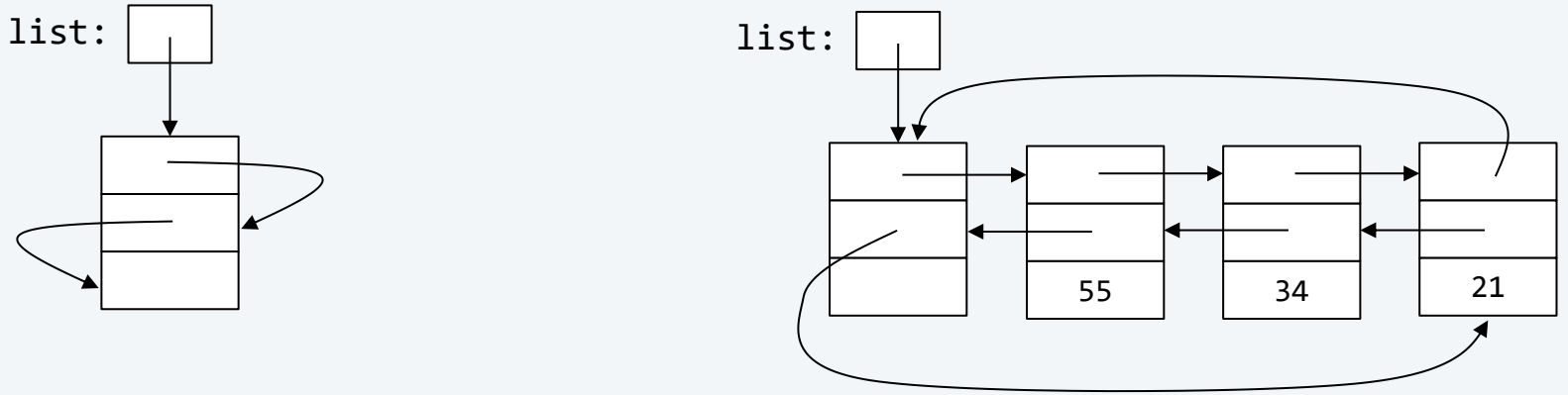
ersten Knoten entfernen

letzten Knoten entfernen

Knoten innerhalb der Liste entfernen

6.6.4 Doppelt-verkettete Liste – als Ringliste realisiert

Implementierung von Listenoperationen kann durch Ringliste und Ankerelement vereinfacht werden



- Das Ankerelement ist ein sogenanntes Dummy-Element (kein echtes Listenelement)
- `list→next` zeigt auf den ersten, `list→prev` auf den letzten Knoten der Liste
- Das Ankerelement wird nicht entfernt

Ringliste – Einfügen eines Knotens in sortierte Ringliste

Algorithmus zum Einfügen eines Knotens in sortierte Ringliste

```
SortedInsert(↓↑list: ListPtr ↓n: ListNodePtr)
  var
    succ: ListNodePtr
begin
  succ := list→next
  while (succ ≠ list) and (n→data > succ→data) do
    succ := succ→next
  end
  n→prev := succ→prev
  n→next := succ
  succ→prev→next := n
  succ→prev := n
end SortedInsert
```

keine Sonderfallbehandlung erforderlich!

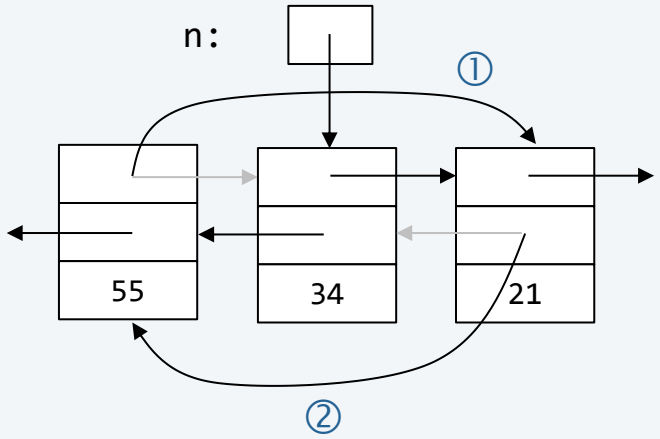
Der Vorteil sortierter Ringlisten zeigt sich hier besonders (drastische Vereinfachung des Algorithmus zur sortierten Einfügung)

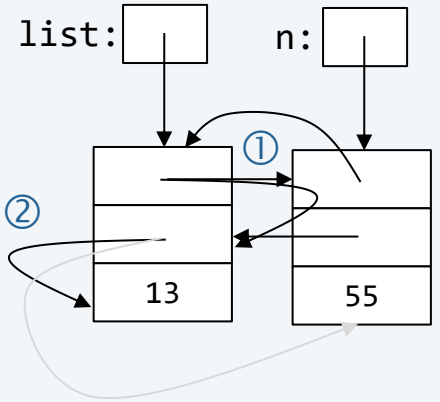
Ringliste – Entfernen eines Knotens

Algorithmus zum Entfernen eines Knotens aus einer Ringliste

```
Delete(↓n: ListNodePtr)
begin
  assert n ∈ list
  n→prev→next := n→next   ①
  n→next→prev := n→prev   ②
  n→prev := null
  n→next := null
  -- ggf. Dispose(↓n)
end Delete
```

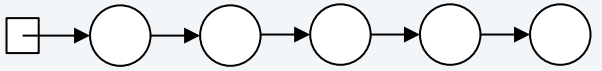
keine Sonderfallbehandlung erforderlich!





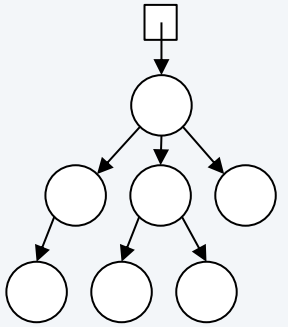
6.7 Bäume – Grundlegendes

Listen sind lineare Datenstrukturen



- Ein Element (Knoten) folgt dem anderen
- Jedes Element hat einen Vorgänger und einen Nachfolger

Bäume sind wichtige nicht-lineare (zweidimensionale) Datenstrukturen

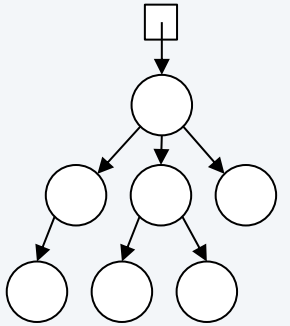


Beispiele:

- Familienstammbaum (jede Person hat einen Vater und eine Mutter)
- Unternehmensstruktur (Konzern, Profitcenter, Abteilung, ...)
- Struktur von Dokumenten (Kapitel, Abschnitte, Paragraphen, ...)

Begriffsdefinition

Ein (allgemeiner) Baum ist entweder leer, oder er repräsentiert eine Menge von Knoten, in der es genau einen ausgezeichneten Knoten, die Wurzel, und eine endliche Menge von disjunkten Bäumen, die sogenannten Teilbäume gibt, die mit der Wurzel verbunden sind.



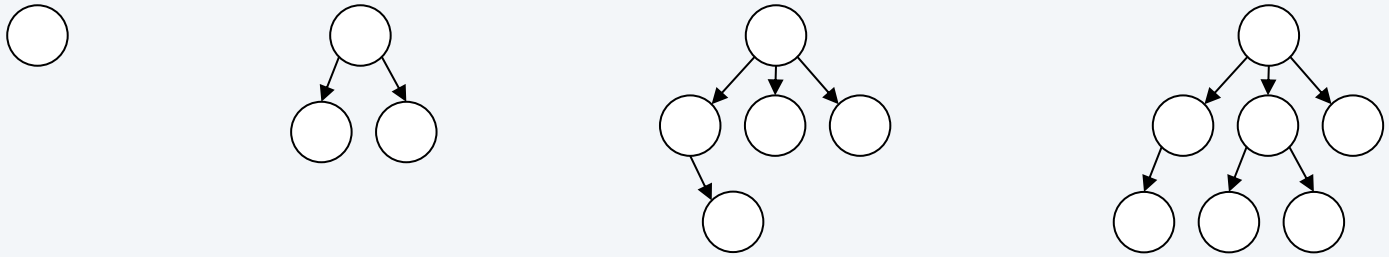
disjunkt: jeder Teilbaum hat keinen Kontakt zu einem anderen Teilbaum - irgendwie so

Aus dieser Definition und ihrem Vergleich mit der Definition für verkettete Listen geht hervor, dass Bäume den verketteten Listen ähnlich sind:

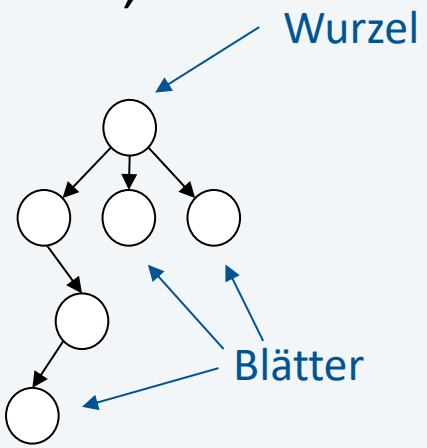
- wie Listen, bestehen auch Bäume aus Knoten
- wie bei Listen hat auch bei Bäumen jeder Knoten höchstens einen Vorgänger
- während in Listen jeder Knoten höchstens einen Nachfolger hat, kann jeder Baumknoten beliebig viele Nachfolger aufweisen

Zur Begriffsdefinition – Eigenschaften von Bäumen

Ein Baum ist aus einer Menge von Knoten und Kanten zusammengesetzt



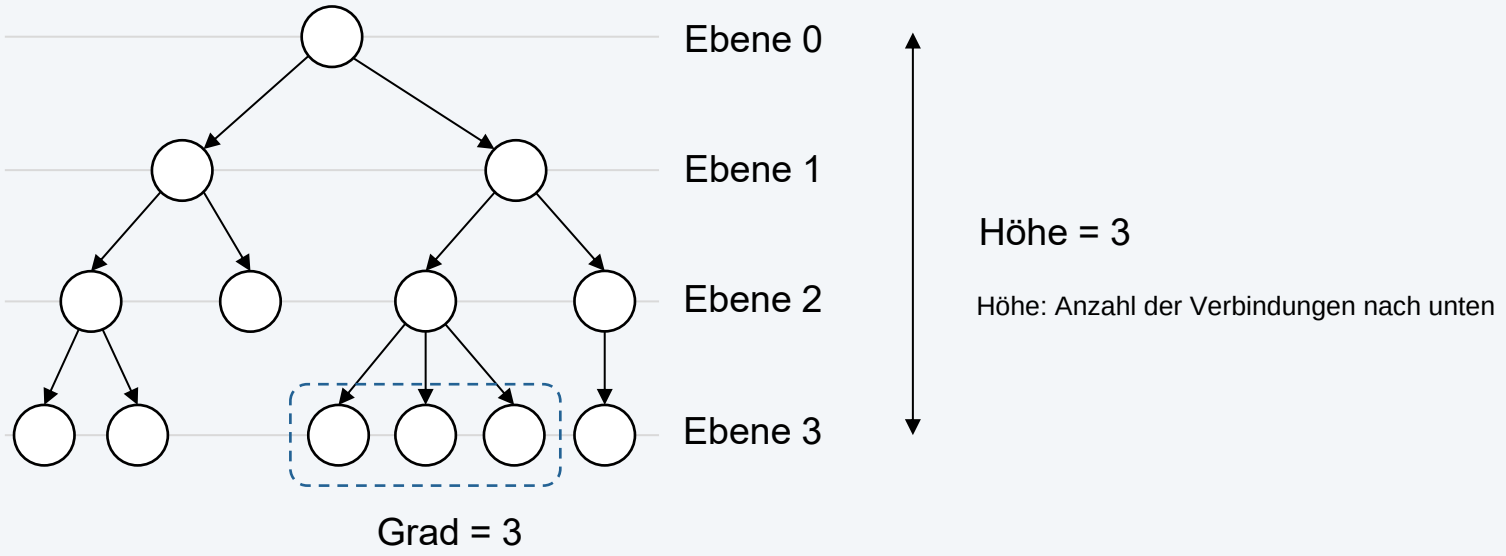
- Auch ein einzelner Knoten kann ein Baum sein
- Jeder Knoten enthält Daten und kann mehrere Nachfolger haben, die selbst wieder Bäume sind
- Knoten ohne Vorgänger nennen wir Wurzel
- Knoten ohne Nachfolger nennen wir Blätter
- Bäume sind rekursive Datenstrukturen, da Nachfolger von Knoten selbst wieder als Wurzel eines Baums betrachtet werden können



Zur Begriffsdefinition – Ebene, Höhe und Grad

Jeder Knoten ist einer bestimmten Ebene im Baum zugeordnet, die Wurzel der Ebene 0

Die Höhe eines Baums wird bestimmt durch die Länge des Wegs, d. h. durch die Anzahl der Verbindungen von der Wurzel bis zu einem Blatt auf der höchsten Ebene



Der Grad eines Baums ist die maximale Anzahl der direkten Nachfolger eines Knotens

6.8 Binärbäume – Begriffsdefinitionen

Ein Binärbaum ist ein spezieller Baum, in dem jeder Knoten maximal zwei Nachfolger hat (Grad = 2)

Es ist üblich, die beiden Nachfolger als linken und rechten Nachfolger, bzw. die in diesen Knoten wurzelnden Teile des Binärbaums als linken und rechten Teilbaum zu bezeichnen

Je nach Anzahl und Position der Knoten im Binärbaum unterscheidet man verschiedene Arten, von denen nur die beiden wichtigsten erwähnt sind:

- ein vollständiger Binärbaum ist ein Baum, in dem jeder Knoten entweder gar keinen oder genau zwei Nachfolger hat
- ein perfekter Binärbaum ist ein vollständiger Binärbaum, in dem alle Blätter auf ein und derselben Ebene liegen, die Anzahl der Knoten eines perfekten Binärbaums beträgt $2^k - 1$ für $k = \text{Höhe} + 1$

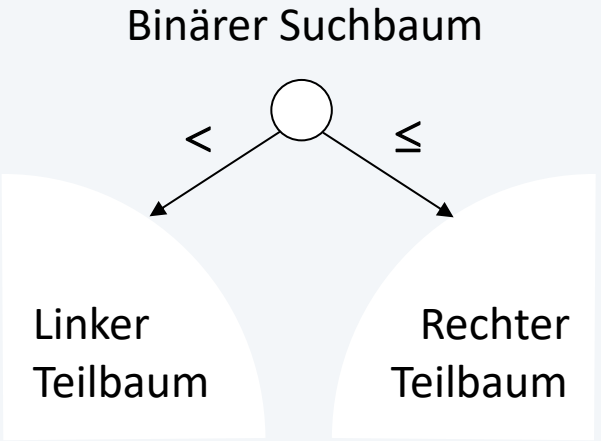
ein nicht vollständiger Binärbaum kann auch nicht perfekt sein.

Suchbäume – Begriffsdefinition und Eigenschaften

Häufigste Verwendungsart von Binärbäumen sind binäre Suchbäume (geordnete Binärbäume)

Ein binärer Suchbaum ist ein Binärbaum, dessen Knoten so angeordnet sind, dass

- die Werte der Datenkomponenten data aller linken Nachfolger des betrachteten Knotens kleiner und
- die aller rechten Nachfolger größer oder gleich als der Wert von data eines betrachteten Knoten sind



Diese Eigenschaft kann zur effizienten Suche und Einfügung eines bestimmten Knoten genutzt werden

Die Datenkomponente nennen wir auch Ordnungsschlüssel

Ausprägungen binärer Suchbäume

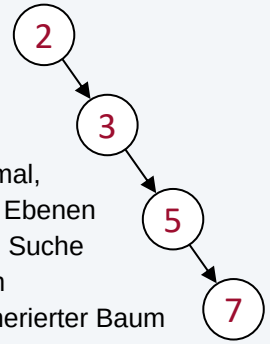
Die Ausprägung eines binären Suchbaums hängt davon ab, in welcher Reihenfolge die Knoten bei der Bildung des Baums eingefügt werden

Die Anzahl der möglichen Ausprägungen eines binären Suchbaums für n darin zu speichernde Werte (und somit Knoten) ist durch die Catalan'sche Zahl $C(n)$ bestimmt:

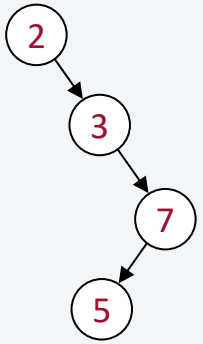
$$C(n) = \frac{1}{n+1}\binom{2n}{n}$$

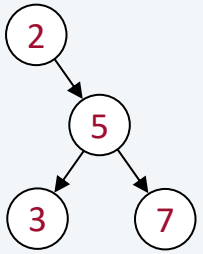
Ausprägungen binärer Suchbäume

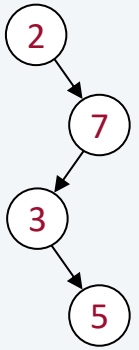
Beispiel: Auswahl möglicher Anordnungen der ersten vier Primzahlen in binären Suchbäumen ($C(4) = 14$)

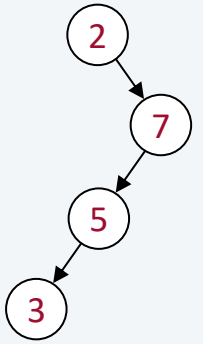


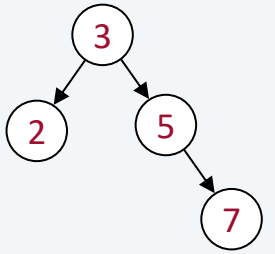
suboptimal,
da viele Ebenen
dadurch Suche
langsam
-- degenerierter Baum

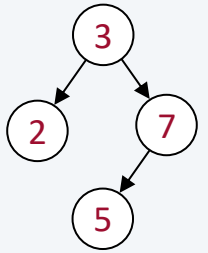


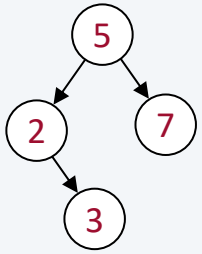


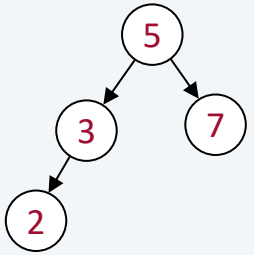


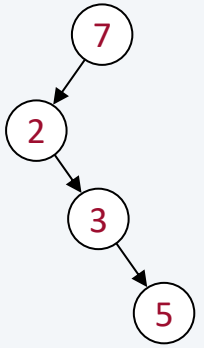


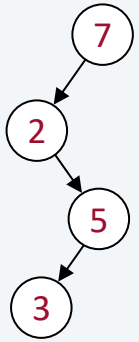


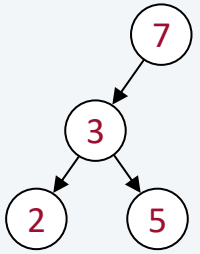


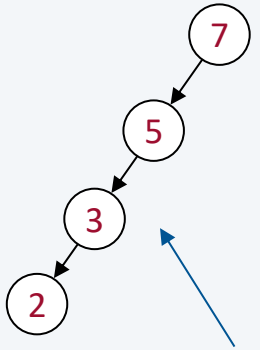


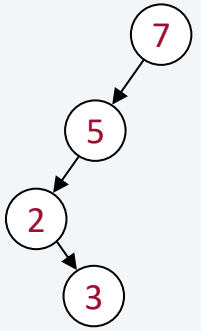










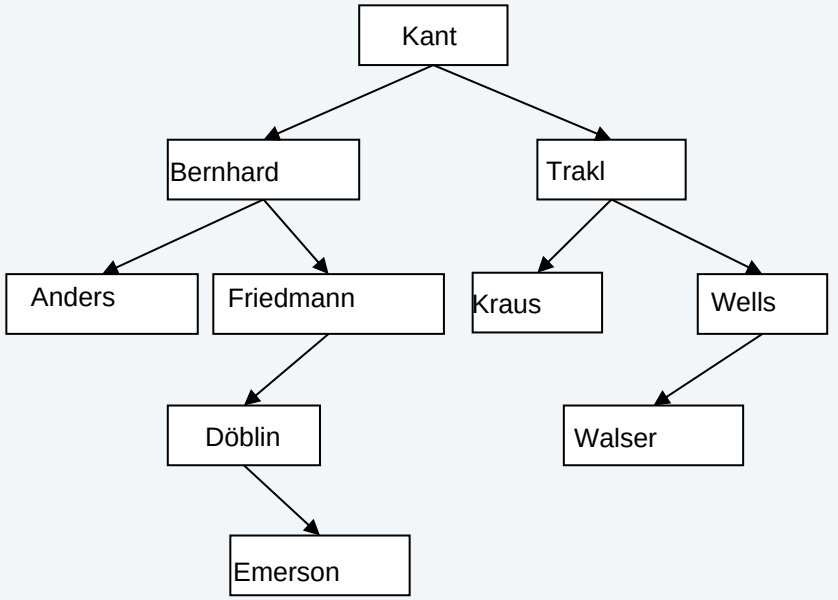


-- Reihenfolge beim Einfügen ist relevant
somit sollte man sich überlegen, in welcher Reihenfolge
die Knoten eingefühgt werden

Degenerierter Baum
(Liste!)

Beispiel eines binären Suchbaumes

Beispiel: Aufbau eines geordneten Binärbaums (Suchbaums) für die folgenden zu speichernden Daten: Kant, Bernhard, Trakl, Friedmann, Döblin, Kraus, Emerson, Anders, Wells, Walser.

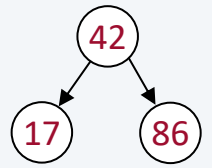


Anders
Bernhard
Döblin
Emerson
Friedmann
Kant
Kraus
Trakl
Walser
Wells

Das Ergebnis hängt von der Reihenfolge ab, in der Daten eingefügt werden (werden die Namen in alphabetischer Reihenfolge eingefügt, entsteht eine Liste, also ein entarteter Suchbaum) (das is also ned so smart)

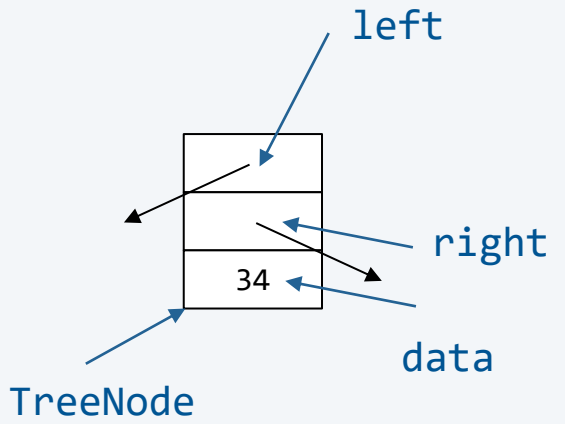
Binärbäume – Deklarationserfordernisse

Erforderliche Deklarationen für einen Binärbaum, dessen (einzige) Datenkomponente beispielsweise vom Typ Integer ist



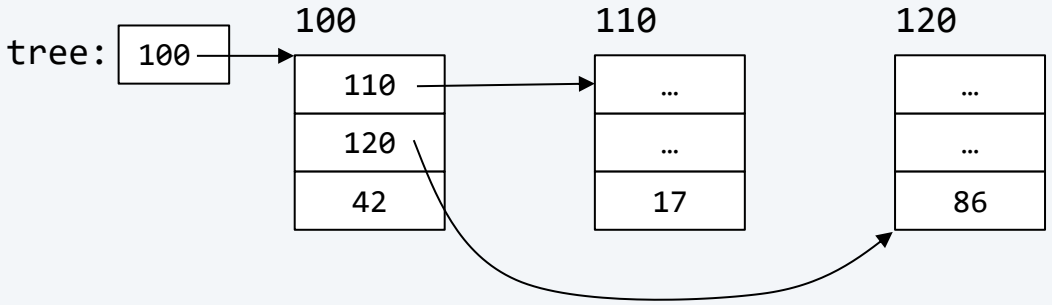

```
type
  TreeNodePtr = →TreeNode
  TreeNode = compound
    left, right: TreeNodePtr
    data: int
  end -- TreeNode
  TreePtr = TreeNodePtr
var
  tree: TreePtr
```

nur für Lesbarkeit
erster Knoten nennen wir TreePtr
um ihn unterscheiden zu können



Erzeugen eines Baums:

```
tree := New(↓TreeNode)
tree→data := 42
tree→left := New(↓TreeNode)
tree→left→data := 17
tree→right := New(↓TreeNode)
tree→right→data := 86
```



Binäre Suchbäume – Suche eines bestimmten Knotens

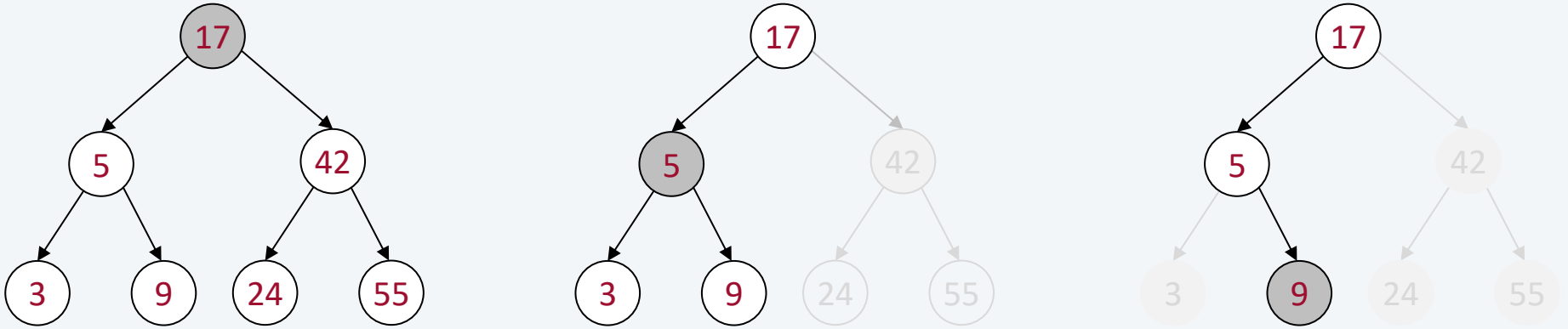
Gegeben sei ein binärer Suchbaum

Gesucht sei ein Algorithmus, der zu einem gegebenen Knoteninhalt x (Ordnungsschlüsselwert) den Zeiger auf diesen Knoten oder `null`, falls ein solcher nicht existiert, liefert

Beispiel: Es sei folgender binäre Suchbaum gegeben, und es soll der Zeiger des Knotens, der den Ordnungsschlüsselwert x = 9 enthält, ermittelt werden

so ein Baum ist perfekt zum binär suchen

deshalb ist ein zumindest annähernd perfekt ist, ist gut



Binäre Suchbäume – Suche eines bestimmten Knotens

Lösungsidee:

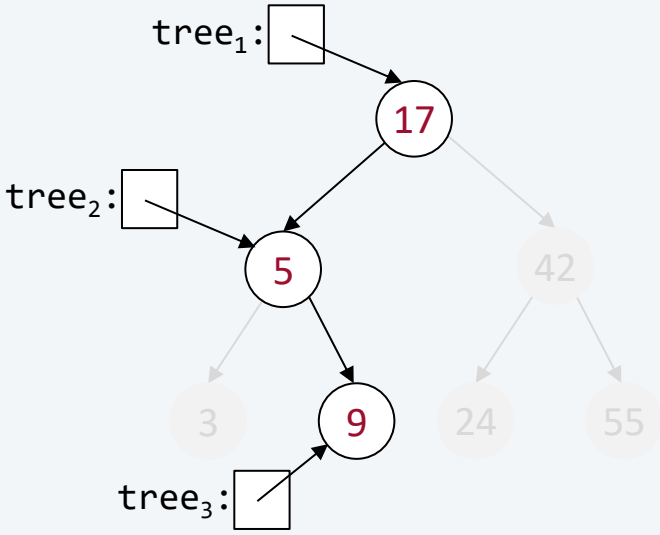
- Wenn Wurzel `null`, d. h. der Suchbaum leer ist,
  liefere das Ergebnis `null` (= nicht gefunden)
- Wenn der gesuchte Wert x bereits im Wurzelknoten enthalten ist,
  liefere als Ergebnis den Zeiger auf den Wurzelknoten
- Wenn der gesuchte Wert x kleiner als Wert des Wurzelknotens ist,
  dann suche im linken Teilbaum -- hier ist die Rekursion
- Wenn der gesuchte Wert x größer als Wert des Wurzelknotens ist,
  dann suche im rechten Teilbaum
- Wenn der gesuchte Wert x in keinem der beiden Teilbäume vorkommt,
  liefere das Ergebnis `null` (= nicht gefunden)

Binäre Suchbäume – Suche eines bestimmten Knotens

Transformation der Lösungsidee in einen Algorithmus:

```
BinTreeSearch(↓tree: TreePtr ↓x: int): TreeNodePtr
begin
  if tree = null then
    return null
  elsif tree→data = x then
    return tree
  elsif x < tree→data then
    return BinTreeSearch(↓tree→left ↓x)
  else
    return BinTreeSearch(↓tree→right ↓x)
  end
end BinTreeSearch
```

endrekursiv (und linear), daher gibts nen einfachen iterativen Algo



Einfacher, eleganter Algorithmus, der allerdings voraussetzt, dass sich ein Algorithmus selbst aufrufen kann.

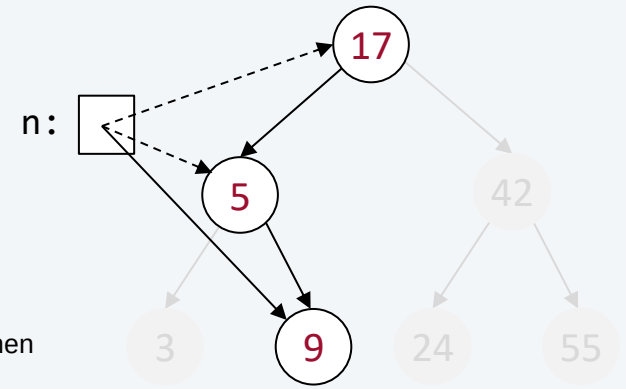
Wir bezeichnen einen solchen Algorithmus als rekursiv (siehe Kapitel 7)

Binäre Suchbäume – Suche eines bestimmten Knotens

Wie könnte ein iterativer Algorithmus dafür konstruiert sein?

Weil die rekursiven Aufrufe nur am Ende des Algorithmus vorkommen, ist ein solcher einfach anzugeben

```
BinTreeSearch(↓tree: TreePtr ↓x: int): TreeNodePtr
  var
    n: TreeNodePtr
begin
  n := tree
  while (n ≠ null) and (n→data ≠ x) do
    if x < n→data then
      n := n→left
    else
      n := n→right
    end
  end
  return n
end BinTreeSearch
```



iterativ ist zwar etwas schneller,
aber bei annähernd perfekten Bäumen
kein Problem
Bei 1023 Knoten sinds beim Perfekten Baum
ja nur 10 Ebenen und somit nur 10 rekursive Aufrufe.

Iterative Lösung entspricht Suche in verketteten Listen:
n := n→next wird durch n := n→left oder n→right ersetzt

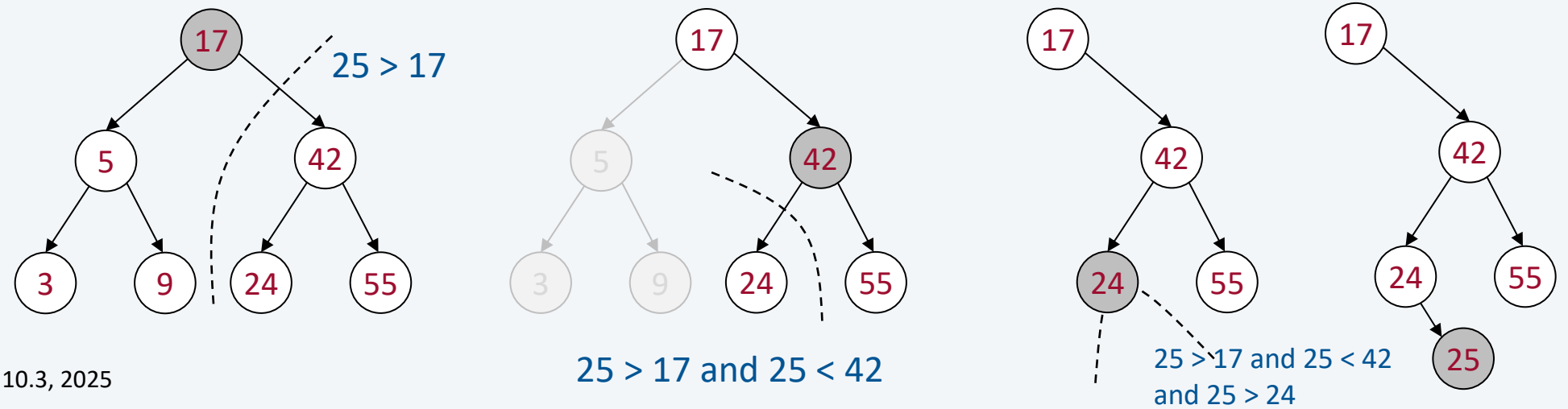
Binäre Suchbäume – Einfügen eines bestimmten Knotens

Gegeben sei ein binärer Suchbaum, d. h. seine Wurzel `tree` und der Zeiger auf einen Knoten `n` (vom Datentyp der Knoten des Suchbaumes)

Gesucht ist ein Algorithmus, der den Knoten `n`, entsprechend des Inhalts seiner Datenkomponente (seines Schlüsselwerts), an der richtigen Stelle, in dem durch `tree` identifizierten binären Suchbaum, einfügt

Annahmen: Der Suchbaum existiert bereits (`tree ≠ null`) und für jeden Schlüsselwert ist nur ein Knoten im Baum vorzusehen

Beispiel: es soll ein Knoten mit dem Schlüsselwert 25 eingefügt werden:



Suchbäume – Einfügen eines bestimmten Knotens

Lösungsidee:

- Prüfen, ob der Schlüsselwert des einzufügenden Knotens (n→`data`) kleiner, gleich oder größer als der des Wurzelknotens (`tree`→`data`) ist
- Wenn er gleich ist, sind wir fertig (es existiert bereits ein Knoten mit dem Schlüsselwert des einzufügenden Knotens und wir haben in der Aufgabenstellung festgelegt, dass für jeden Schlüsselwert nur ein Knoten im Baum vorzusehen ist) -- je nach Aufgabensstellung: hier sollen Werte nur einmal vorkommen soll
- Wenn er kleiner ist, prüfen wir ob im Wurzelknoten bereits ein linker Teilbaum verankert ist; wenn ja, dann suchen wir die Einfügestelle im linken Teilbaum, sonst haben wir die Einfügestelle gefunden und können den Knoten n einfügen
- Wenn er größer ist, verfahren wir analog, d. h. wir prüfen ob im Wurzelknoten bereits ein rechter Teilbaum verankert ist; wenn ja, dann suchen wir die Einfügestelle im rechten Teilbaum, sonst haben wir die Einfügestelle gefunden und können den Knoten n einfügen

Binäre Suchbäume – Einfügen eines bestimmten Knotens

Transformation der Lösungsidee in einen Algorithmus:

```
BinTreeInsert(↓tree: TreePtr ↓n: TreeNodePtr)
begin
   if n→data = tree→data then
     -- node exists (root)
   elsif n→data < tree→data then
     if tree→left = null then
       tree→left := n
     else
       BinTreeInsert(↓tree→left ↓n)
     end
   else -- n→data > tree→data
     if tree→right = null then
       tree→right := n
     else
       BinTreeInsert(↓tree→right ↓n)
     end
   end
end BinTreeInsert
```

Einfügestelle gefunden

Einfügestelle gefunden

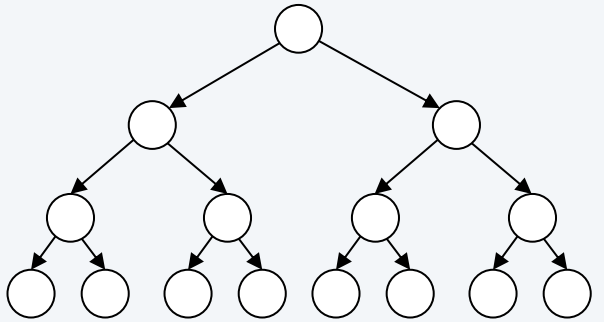
Binäre Suchbäume – Eigenschaften

Die Anzahl der benötigten Vergleiche, um einen Knoten zu finden, hängt von der Höhe des Baums ab

Ein Baum der Höhe h hat, wenn er perfekt ist,

- 2^h Blätter
- $2^h - 1$ innere Knoten
- $2^{h+1} - 1$ Knoten insgesamt

Beispiel: Höhe $h = 3$



Daraus folgt die Höhe h aus Anzahl der Knoten n

$$h = \lceil ld(n+1) \rceil - 1$$

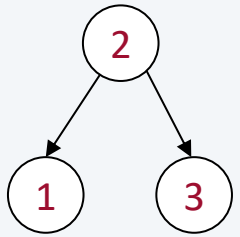
ld: Zweierlog (ln)

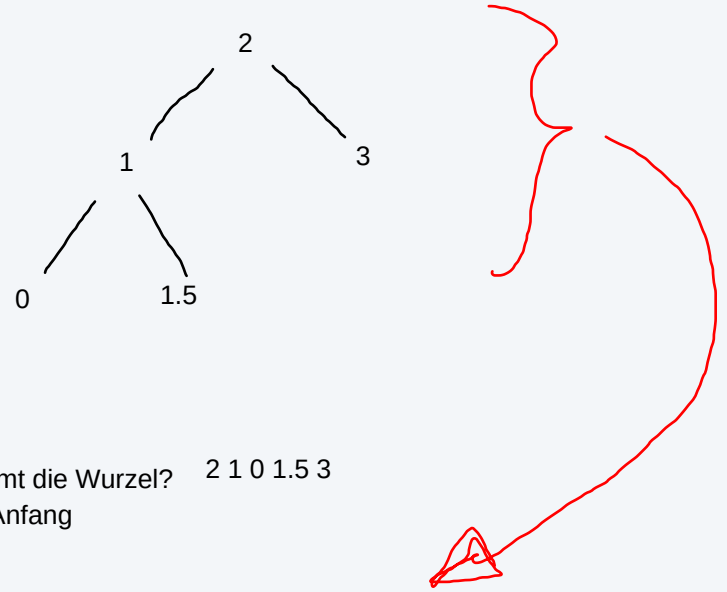
Bei n Knoten sind maximal $\approx ld(n)$ Vergleiche notwendig, um einen Knoten zu suchen oder einzufügen

Durchwandern (Durchlaufen) von Binärbäumen

Oft müssen alle Knoten eines Baums systematisch in einer bestimmten Reihenfolge durchwandert werden

Dazu können wir drei Reihenfolgen unterscheiden:
Prä-Order, In-Order und Post-Order





Damit sind folgende Reihenfolgen verbunden:

- Prä-Order: 2, 1, 3
- In-Order: 1, 2, 3
- Post-Order: 1, 3, 2

Vorsilbe Prä, In, Post: Wo kommt die Wurzel? 2 1 0 1.5 3
Prä-Order: Wurzel kommt am Anfang

0 1 1.5 2 3

0 1.5 1 3 2

Algorithmen für Prä-, In- und Post-Order-Durchlauf

Algorithmus

```
PreOrder(↓tree: TreePtr)
begin
  if tree ≠ null then
    -- process node tree
    PreOrder(↓tree→left)
    PreOrder(↓tree→right)
  end
end PreOrder
```

nicht-linear rekursiv - also iterativ nicht so einfach
mit allgemeinem Schema würds natürlich trotzdem immer gehen

solange die Bäume annähernd perfekt sind, spricht nichts wirklich gegen die Anwendung der Rekursion --> viel einfacher und nicht viele Ebenen

Mit dem Algorithmus `PreOrder` wird zuerst der Wurzelknoten, dann der linke und danach der rechte Teilbaum bearbeitet

Beispiel: So muss z. B. ein Binärbaum, in dem die Daten der Baugruppen eines technischen Produkts gespeichert sind, durchlaufen werden, wenn man zuerst die einer Baugruppe gemeinsamen Daten und danach die Daten ihrer Teile bearbeiten möchte

Algorithmen für *Prä-*, *In-* und *Post-Order*-Durchlauf

Wenn wir einen Stack zur Zwischenspeicherung von Baumknoten verwenden, lässt sich folgender iterative Algorithmus konstruieren:

```
PreOrderIterativ(↓tree: TreePtr)
begin
  InitStack()
  while tree ≠ null do
    -- process node tree
    if tree→right ≠ null then
      Push(↓tree→right)
    end
    if tree→left ≠ null then
      tree := tree→left
    else
      if not IsEmpty() then
        Pop(↑tree)
      else
        tree := null
      end
    end
  end
end PreOrderIterativ
```

Bevor man links absteigt rechten Teilbaum merken

Abstieg in linken Teilbaum

Weiter beim zuletzt gemerkten rechten Teilbaum

```
interface TreePtrStack
  InitStack()
  Push(↓t: TreePtr)
  Pop(↑t: TreePtr)
  IsEmpty(): bool
end TreePtrStack
```

Algorithmen für *Prä-*, *In-* und *Post-Order*-Durchlauf

Algorithmus:

```
InOrder(↓tree: TreePtr)
begin
  if tree ≠ null then
    InOrder(↓tree→left)
    -- process node tree
    InOrder(↓tree→right)
  end
end InOrder
```

wenn in einem Suchbaum absteigend sortiert ausgegeben wird,
auch inOrder aber die markierten Aufrufe (rot) vertauscht

Mit dem Algorithmus `InOrder` wird zuerst der linke Teilbaum, dann der Wurzelknoten und zum Schluss der rechte Teilbaum bearbeitet

Beispiel: So muss z. B. ein binärer Suchbaum durchlaufen werden, wenn man die Inhalte der Datenkomponenten in aufsteigend sortierter Reihenfolge ausgeben möchte

Algorithmen für *Prä-*, *In-* und *Post-Order*-Durchlauf

Algorithmus:

```
PostOrder(↓tree: TreePtr)
begin
  if tree ≠ null then
    PostOrder(↓tree→left)
    PostOrder(↓tree→right)
    -- process node tree
  end
end PostOrder
```

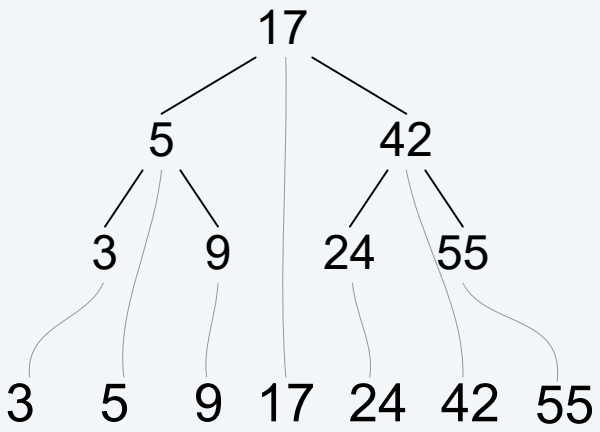
Beispiel: Ein Compailer berechnet 3+4*5

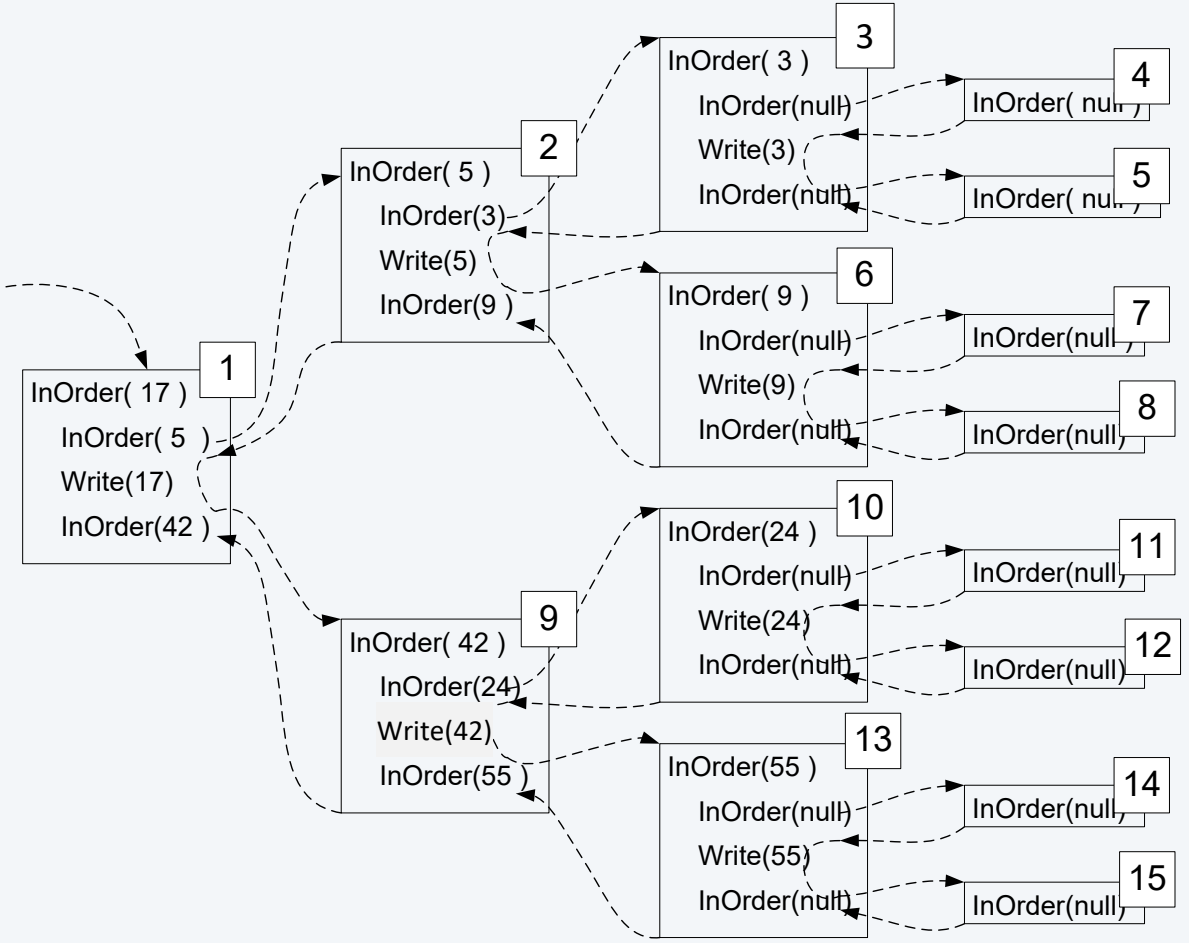
Mit dem Algorithmus `PostOrder` wird zuerst der linke, dann der rechte Teilbaum und erst zum Schluss der Wurzelknoten bearbeitet

Beispiel: So muss z. B. ein binärer Suchbaum, in dem die Operanden und Operatoren eines arithmetischen Ausdrucks gespeichert sind, durchlaufen werden, wenn man den Ausdruck auswerten, d. h. sein Ergebnis ermitteln möchte

Beispiel für In-Order-Durchlauf – sortierte Ausgabe

Folgender binärer Suchbaum soll In-Order durchwandert werden und dabei der Inhalt des jeweiligen Knoten (Integer-Wert) ausgegeben werden





Beispiel für Post-Order-Durchlauf

Beispiel: Entfernen aller Baumknoten

- Bevor ein Knoten entfernt werden darf, müssen alle Knoten des linken und rechten Teilbaums entfernt sein
- Zuerst müssen also die Blätter entfernt werden
- Erst zuletzt darf die Wurzel entfernt werden --sonst kein Zeiger mehr auf die anderen - oder so

Algorithmus:

```
DeleteBinTree(↓tree: TreePtr)
begin
  if tree ≠ null then
    DeleteBinTree(↓tree→left)
    DeleteBinTree(↓tree→right)
    Dispose(↓tree)
  end
end DeleteBinTree
```

Post-Order, da die Wurzel erst nach den Teilbäumen entfernt werden darf (sonst sind diese nicht mehr verwendbar)

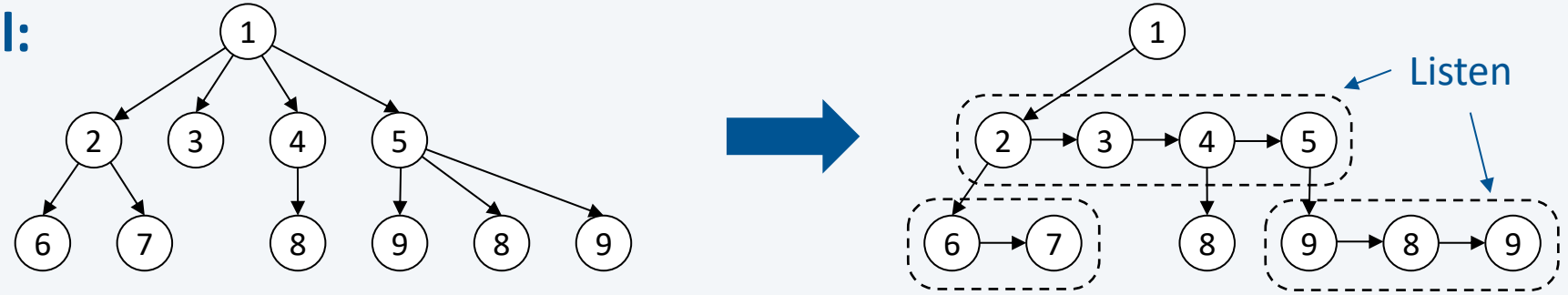
6.9 Allgemeine Bäume

Knoten allgemeiner Bäume haben beliebig viele Nachfolgerknoten – das hat den Nachteil, dass die Datentypen für die Knoten (wegen unterschiedlich vieler Zeiger auf Nachfolger) verschieden sein müssten

Lösung durch Anordnung der Nachfolger in linearer Liste

triviale Lösung wäre, einfach im Parentknoten alle Zeiger auf die Nachfolger in einer Liste zu speichern
jedoch braucht man viel mehr Pointer als wenn man die Knoten direkt in einer Liste speichert.
Dann hat man den allgemeinen Baum in einen Binärbaum transformiert

Beispiel:



Allgemeiner Baum

Binärbaum

Jeder allgemeine Baum kann also in einen Binärbaum transformiert werden

Beispiel für Prä-Order-Durchlauf

Beispiel: Ausgabe Inhaltsverzeichnis

Gegeben seien folgende Deklarationen (zur Verwaltung eines Texts der Teile eines Buchs, z. B. Kapitel, enthält)

```
type
  BookPartPtr = →BookPart
  BookPart = compound
     headLine: string
     ...
     firstSon: BookPartPtr
     nextBrother: BookPartPtr
   end -- BookPart

var
  book: BookPartPtr
```

Beispiel für Prä-Order-Durchlauf

Algorithmus:

```
PrintTOC(↓part: BookPartPtr)
  var
    sub: BookPartPtr
begin
  if part ≠ null then
    WriteLn(↓part→headLine)
    sub := part→firstSon
    while sub ≠ null do
      PrintTOC(↓sub)
      sub := sub→nextBrother
    end
  end
end PrintTOC
```

```
PrintTOC(↓part: BookPartPtr)
  var
    sub: BookPartPtr
begin
  if part ≠ null then
    WriteLn(↓part→headLine)
    PrintTOC(↓part→firstSon)
    PrintTOC(↓part→nextBrother)
  end
end PrintTOC
```