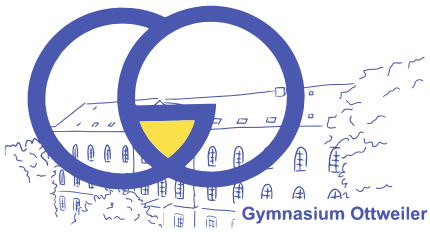
Kapitel 3

Rekursion und dynamische Datenstrukturen



Skript Informatik · Grundkurs

Dieses Kapitel bringt zwei Dinge zusammen, die zusammengehören. Zuerst die **Rekursion** — ein Verfahren, das sich selbst aufruft. Dann die **dynamischen Datenstrukturen**, deren Größe erst zur Laufzeit feststeht: verkettete Listen und binäre Bäume.

Dass beides in einem Kapitel steht, hat einen Grund. Ein Baum besteht aus Bäumen: Jeder Knoten trägt links einen Baum und rechts einen Baum. Eine Struktur, die so gebaut ist, will mit einem Verfahren bearbeitet werden, das genauso gebaut ist. Wer die Rekursion beherrscht, schreibt die Algorithmen auf Bäumen in drei Zeilen; wer es mit Schleifen versucht, kämpft.

Lernziele

Am Ende dieses Kapitels können Sie

- die Merkmale rekursiver Methoden und Definitionen benennen,
- lineare und baumförmige Rekursion unterscheiden und ihre Aufrufschemata skizzieren,
- die Fakultät und die Fibonacci-Zahlen rekursiv definieren und implementieren,
- rekursive und iterative Fassungen ineinander umwandeln,
- Vor- und Nachteile rekursiver Lösungen beurteilen,
- eine verkettete Liste aufbauen, durchlaufen und erweitern,
- binäre Bäume definieren, grafisch darstellen und die Begriffe Wurzel, Teilbaum, Knoten, Blatt und Tiefe erläutern,
- Knotenzahl und Tiefe eines Baums rekursiv bestimmen,
- die drei Durchlaufarten preorder, inorder und postorder ausführen,
- in einem Suchbaum suchen, einfügen und löschen.

3.1 Rekursion

Versuch: Der Stapel Umschläge

Auf dem Pult liegt ein Umschlag. Darin: ein Zettel mit einer Zahl — und ein weiterer Umschlag. Darin wieder eine Zahl und ein Umschlag. Und so fort, bis irgendwann ein Umschlag kommt, der nur eine Zahl enthält und sonst nichts.

Aufgabe: Bestimmen Sie die Summe aller Zahlen. Eine Person darf immer nur **einen** Umschlag in der Hand halten.

Auswertung

Es gibt nur ein Vorgehen, das funktioniert: Sie lesen Ihre Zahl, geben den inneren Umschlag an den Nächsten weiter und warten. Irgendwann kommt von dort eine Summe zurück; Sie zählen Ihre Zahl dazu und geben das Ergebnis nach vorn.

Zwei Dinge fallen auf. Erstens hat niemand den ganzen Stapel überblickt — jeder kannte nur seinen Umschlag und die Regel. Zweitens ging es nur gut, weil es einen letzten Umschlag *ohne* Inhalt gab. Ohne ihn stünden Sie noch dort.

Definition: Rekursion

Ein Unterprogramm heißt **rekursiv**, wenn es sich selbst aufruft. Jede rekursive Lösung braucht zwei Teile:

- den **Basisfall** (Rekursionsanfang): der Fall, der ohne weiteren Aufruf beantwortet wird;
- den **Rekursionsschritt**: die Rückführung des Problems auf ein *kleineres* Problem derselben Art.

Gymnasium Ottweiler

Merke

Fehlt der Basisfall, oder wird das Problem nicht wirklich kleiner, läuft die Rekursion endlos — bis der Speicher für die offenen Aufrufe voll ist. Das Programm stürzt dann mit einer Meldung wie `RecursionError` oder `Stack overflow` ab.

Prüfen Sie bei jeder rekursiven Methode beides: **Gibt es einen Basisfall? Und führt jeder Schritt sicher dorthin?**

3.1.1 Die Fakultät

Die Fakultät kennen Sie als Produkt: $n! = 1 \cdot 2 \cdot \ldots \cdot n$. Rekursiv geschrieben sieht sie so aus:

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

Die obere Zeile ist der Basisfall, die untere der Rekursionsschritt.

Python

```python
def fakultaet(n):
    if n == 0:
        return 1
    return n * fakultaet(n - 1)
```

Object Pascal

```
function Fakultaet(n: Integer): Int64;
begin
  if n = 0 then
    Result := 1
  else
    Result := n * Fakultaet(n - 1);
end;
```

So entfaltet sich `fakultaet(4)`:

```
fakultaet(4) = 4 * fakultaet(3)
                   fakultaet(3) = 3 * fakultaet(2)
                                      fakultaet(2) = 2 * fakultaet(1)
                                                         fakultaet(1) = 1 *
                                                         fakultaet(0)
                                                                          fakultaet
                                                                          (0) =
                                                                          1
                                                         fakultaet(1) = 1
                                      fakultaet(2) = 2
                   fakultaet(3) = 6
fakultaet(4) = 24
```

Gut zu sehen: Erst geht es nach unten bis zum Basisfall, dann rechnen sich die Werte wieder nach oben. Nichts wird berechnet, bevor der Basisfall erreicht ist.

3.1.2 Lineare und baumförmige Rekursion

Definition: Zwei Aufrufschemata

Bei der **linearen Rekursion** ruft sich die Methode pro Durchlauf **einmal** selbst auf. Die Aufrufe bilden eine Kette.

Bei der **baumförmigen Rekursion** ruft sie sich **mehrfach** selbst auf. Die Aufrufe bilden einen Baum.

Die Fakultät ist linear — eine Kette:

```
fak(4) → fak(3) → fak(2) → fak(1) → fak(0)
```

Die Fibonacci-Zahlen sind baumförmig. Sie sind definiert durch

$$\text{fib}(n) = \begin{cases} n & \text{falls } n < 2 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{falls } n \geq 2 \end{cases}$$

und ergeben die Folge 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

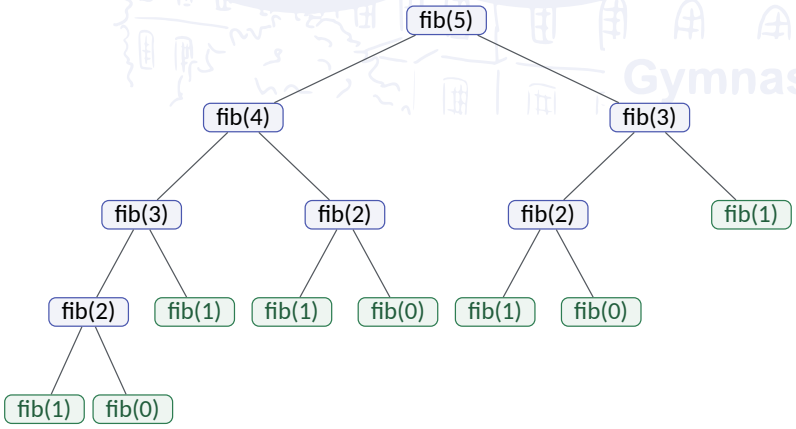
Python

```python
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

Object Pascal

```
function Fib(n: Integer): Int64;
begin
  if n < 2 then
    Result := n
  else
    Result := Fib(n - 1) + Fib(n - 2);
end;
```

Der Aufrufbaum für `fib(5)`:



15 Aufrufe für ein Ergebnis von 5. Grün = Basisfall. `fib(2)` wird dreimal gerechnet, `fib(1)` fünfmal.

Aufrufbaum von `fib(5)`.

> In der Webfassung steht hier ein Bedienfeld zum Ausprobieren: **Rekursionsbaum**.

Beispiel: Was baumförmige Rekursion kostet

Zählt man die Aufrufe, wird das Problem sichtbar:

	`fib(5)`	`fib(10)`	`fib(20)`	`fib(30)`
Ergebnis	5	55	6 765	832 040
Aufrufe	15	177	21 891	2 692 537

`fib(2)` wird beim Aufruf von `fib(5)` dreimal berechnet, `fib(1)` sogar fünfmal — immer mit demselben Ergebnis. Der Baum rechnet dieselben Teilprobleme wieder und wieder.

Stolperstein

Baumförmige Rekursion ist elegant und kann sehr teuer sein. Lassen Sie `fib(50)` nicht auf einem Schulrechner laufen — die Aufrufzahl wächst exponentiell.

Arbeitsauftrag 1

1. Skizzieren Sie das Aufrufschema von `fakultaet(5)` und von `fib(4)`. Welches ist linear, welches baumförmig?
2. Bauen Sie in `fib` einen Zähler ein, der jeden Aufruf mitzählt, und prüfen Sie die Zahlen aus der Tabelle nach.
3. Die Türme von Hanoi: n Scheiben sollen von Stapel A nach C, Stapel B ist Hilfsstapel. Es darf nie eine größere auf einer kleineren liegen. Formulieren Sie Basisfall und Rekursionsschritt in Worten.

3.1.3 Rekursiv oder iterativ?

Jede rekursive Lösung lässt sich auch mit einer Schleife schreiben — und umgekehrt. Bei der Fakultät ist beides kurz:

```
def fakultaet_iterativ(n):
    ergebnis = 1
    for zahl in range(2, n + 1):
        ergebnis = ergebnis * zahl
    return ergebnis
```

Bei Fibonacci lohnt der Wechsel besonders, weil die iterative Fassung sich die beiden letzten Werte merkt, statt sie neu zu berechnen:

```
def fib_iterativ(n):
    vorletzter, letzter = 0, 1
    if n < 2:
        return n
    for _ in range(n - 1):
        vorletzter, letzter = letzter, vorletzter + letzter
    return letzter
```

```
function FibIterativ(n: Integer): Int64;
var
  vorletzter, letzter, hilf, i: Int64;
begin
  if n < 2 then
  begin
    Result := n;
    Exit;
  end;
  vorletzter := 0;
  letzter := 1;
  for i := 2 to n do
  begin
    hilf := vorletzter + letzter;
    vorletzter := letzter;
    letzter := hilf;
  end;
  Result := letzter;
end;
```

Diese Fassung braucht für `fib(30)` genau 29 Durchläufe statt 2 692 537 Aufrufe.

Merke

Vorteile der Rekursion: Sie bildet rekursiv definierte Probleme unmittelbar ab. Der Quelltext ist kurz und steht nah an der mathematischen Definition. Bei Bäumen ist sie praktisch die einzige vernünftige Wahl.

Nachteile: Jeder offene Aufruf belegt Speicher. Die Tiefe ist begrenzt. Und bei baumförmiger Rekursion werden Teilprobleme vielfach doppelt gerechnet.

Faustregel: Ist die Struktur rekursiv (Baum, Umschlag im Umschlag), nehmen Sie Rekursion. Ist sie linear (Feld, Liste, Zählschleife), nehmen Sie eine Schleife.

Arbeitsauftrag 2

4. Schreiben Sie `summe(a, b)` aus Kapitel 1 rekursiv.
5. Wandeln Sie diese iterative Methode in eine rekursive um. *(Die ganzzahlige Potenz steht nur im LK-Lehrplan, dort zusätzlich als `square-and-multiply`.)*

```
def potenz(basis, exponent):
    ergebnis = 1
    for _ in range(exponent):
        ergebnis = ergebnis * basis
    return ergebnis
```

6. Schreiben Sie eine rekursive Methode, die prüft, ob ein Wort ein Palindrom ist (`RENTNER`). Basisfall?

3.2 Verkettete Listen

LK

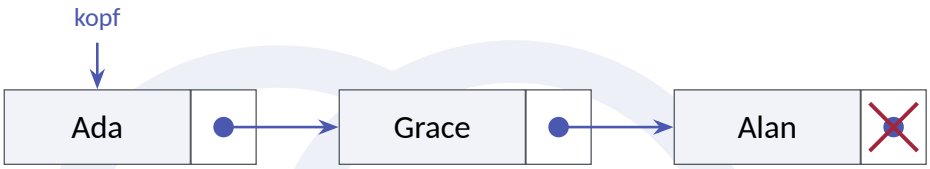
> LK
>
> Lineare Listen stehen nur im Lehrplan des Leistungskurses. Für den Grundkurs sind sie nicht verbindlich — als Vorübung zu den Bäumen aber nützlich, denn das Prinzip „Knoten mit Verweis" ist dasselbe.

Ein Feld hat eine feste Größe. Wer beim Programmieren `array [1..100]` schreibt, hat sich auf hundert Einträge festgelegt — der 101. passt nicht mehr hinein, und bei drei Einträgen liegen 97 Plätze brach.

Definition: Verkettete Liste

Eine **verkettete Liste** besteht aus **Knoten**. Jeder Knoten enthält einen Inhalt und einen **Verweis** auf den nächsten Knoten. Der letzte Knoten verweist auf *nichts*.

Ein Verweis auf den ersten Knoten heißt **Kopf** der Liste. Eine leere Liste ist eine, deren Kopf auf nichts verweist.



Jeder Knoten trägt einen Inhalt und einen Verweis auf den nächsten. Der letzte verweist auf nichts (`nil` bzw. `None`).

Eine verkettete Liste aus drei Knoten.

Die Knoten können im Speicher beliebig weit auseinanderliegen; zusammengehalten werden sie allein durch die Verweise. Dadurch wächst und schrumpft die Liste zur Laufzeit, ganz nach Bedarf.

Object Pascal — hier wird mit Zeigern gearbeitet:

```
type
  TListenZeiger = ^TListenKnoten;
  TListenKnoten = record
    inhalt: String;
    next: TListenZeiger;
  end;
```

Das Zeichen ^ steht für „zeigt auf". Bemerkenswert: `TListenZeiger` benutzt `TListenKnoten`, bevor dieser deklariert ist — bei Zeigertypen ist das ausnahmsweise erlaubt.

Python — dort ist jede Objektvariable ohnehin ein Verweis:

```
class ListenKnoten:
    def __init__(self, inhalt):
        self.inhalt = inhalt
        self.next = None
```

Stolperstein

Ab Abschnitt 3.3 kommt ein **zweiter** Knotentyp dazu: der Baumknoten mit *zwei* Verweisen. Beide heißen umgangssprachlich „Knoten", sind aber verschiedene Typen. Im Skript tragen sie deshalb verschiedene Namen.

3.2.1 Vorn einfügen

Am einfachsten wächst eine Liste am Kopf:

```python
def vorn_einfuegen(kopf, inhalt):
    neu = ListenKnoten(inhalt)
    neu.next = kopf          # der alte Kopf hängt sich hinten an
    return neu               # der neue Knoten ist jetzt Kopf
```

```
procedure VornEinfuegen(var kopf: TListenZeiger; inhalt: String);
var
  neu: TListenZeiger;
begin
  New(neu);
  neu^.inhalt := inhalt;
  neu^.next := kopf;
  kopf := neu;
end;
```

Gymnasium Ottweiler

Stolperstein

In der Pascal-Fassung ist die Reihenfolge der beiden letzten Zeilen entscheidend: Setzen Sie zuerst `kopf := neu`, ist der alte Kopf verloren — und mit ihm die ganze restliche Liste. **Erst anhängen, dann umhängen.**

Die Python-Fassung umgeht das, indem sie den neuen Kopf *zurückgibt*, statt ihn zu ändern. Der Aufrufer schreibt `kopf = vorn_einfuegen(kopf, "Ada")` — vergisst er die Zuweisung, passiert nichts.

3.2.2 Durchlaufen

```python
def ausgeben(kopf):
    aktuell = kopf
    while aktuell is not None:
        print(aktuell.inhalt)
        aktuell = aktuell.next
```

```
procedure Ausgeben(kopf: TListenZeiger);
var
  aktuell: TListenZeiger;
begin
  aktuell := kopf;
  while aktuell <> nil do
  begin
    WriteLn(aktuell^.inhalt);
    aktuell := aktuell^.next;
  end;
end;
```

Der Preis der Beweglichkeit zeigt sich hier: Um an den fünften Eintrag zu kommen, muss man vier Knoten abschreiten. Ein Feld liefert `feld[5]` sofort. **Die Liste kann wachsen, das Feld kann springen** — mehr ist der Unterschied nicht, und er entscheidet, welches passt.

Arbeitsauftrag 3

7. Schreiben Sie eine Anfrage `laenge(kopf)`, die die Zahl der Knoten liefert — einmal mit Schleife, einmal rekursiv.
8. Schreiben Sie `hinten_anhaengen(kopf, inhalt)`. Welchen Sonderfall müssen Sie zuerst behandeln?
9. Warum ist das Einfügen *vorn* billiger als das Einfügen *hinten*? Antworten Sie, ohne zu programmieren.

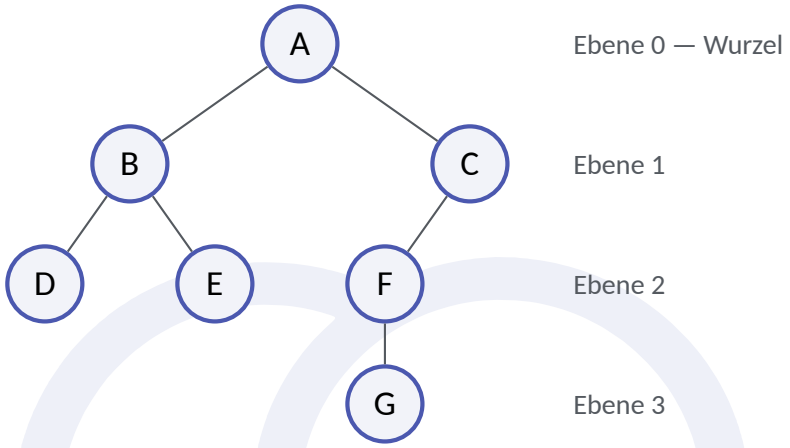
3.3 Binäre Bäume

Eine Liste verzweigt sich nie — jeder Knoten hat einen Nachfolger. Erlaubt man **zwei**, entsteht ein Baum.

Definition: Binärer Baum

Ein **binärer Baum** ist entweder leer, oder er besteht aus einem **Knoten** mit einem Inhalt und zwei binären Bäumen: dem **linken** und dem **rechten Teilbaum**.

Lesen Sie diese Definition zweimal. Sie erklärt den Baum durch Bäume — sie ist selbst rekursiv. Genau deshalb sind die Algorithmen darauf rekursiv.



F ist das *linke* Kind von C, G das *linke* Kind von F. Blätter: D, E, G. Tiefe 4, sieben Knoten.

Ein binärer Baum mit sieben Knoten.

Definition: Grundbegriffe

- **Wurzel** — der oberste Knoten, der einzige ohne Vorgänger.
- **Vorgänger** (Vater) — der Knoten unmittelbar darüber; jeder Knoten außer der Wurzel hat genau einen.
- **Nachfolger** (Kinder) — die Knoten unmittelbar darunter, höchstens zwei.
- **Blatt** — ein Knoten ohne Nachfolger (oben: D, E, G).
- **innerer Knoten** — ein Knoten mit mindestens einem Nachfolger.
- **Teilbaum** — jeder Knoten ist Wurzel des Baums, der unter ihm hängt.
- **Tiefe** (Höhe) — die Zahl der Ebenen. Der Baum oben hat die Tiefe 4.

3.3.1 Knotenzahl und Tiefe

Beide Aufgaben werden in drei Zeilen gelöst, wenn man der Struktur folgt.

Python

```python
class BaumKnoten:
    def __init__(self, inhalt):
        self.inhalt = inhalt
        self.links = None
        self.rechts = None

def anzahl(knoten):
    if knoten is None:
        return 0
    return 1 + anzahl(knoten.links) + anzahl(knoten.rechts)

def tiefe(knoten):
    if knoten is None:
        return 0
    return 1 + max(tiefe(knoten.links), tiefe(knoten.rechts))
```

Object Pascal

```
type
  TBaumZeiger = ^TBaumKnoten;
  TBaumKnoten = record
    inhalt: Integer;
    links, rechts: TBaumZeiger;
  end;

function Anzahl(knoten: TBaumZeiger): Integer;
begin
  if knoten = nil then
    Result := 0
  else
    Result := 1 + Anzahl(knoten^.links) + Anzahl(knoten^.rechts);
end;

function Tiefe(knoten: TBaumZeiger): Integer;
var
  l, r: Integer;
begin
  if knoten = nil then
    Result := 0
  else
  begin
    l := Tiefe(knoten^.links);
    r := Tiefe(knoten^.rechts);
    if l > r then Result := 1 + l else Result := 1 + r;
  end;
end;
```

Der leere Baum ist der Basisfall. Der Rekursionsschritt stellt dieselbe Frage an die beiden Teilbäume — die kleiner sind, also sicher irgendwann leer. Für den Baum oben liefert `anzahl` den Wert 7 und `tiefe` den Wert 4.

Nach demselben Muster wird gesucht. Weil der Baum hier noch *ungeordnet* ist, muss die Methode beide Teilbäume durchsehen — gefunden ist der Wert, wenn er in einem von beiden steckt:

```python
def enthaelt(knoten, wert):
    if knoten is None:
        return False
    if knoten.inhalt == wert:
        return True
    return enthaelt(knoten.links, wert) or enthaelt(knoten.rechts, wert)
```

```pascal
function Enthaelt(knoten: TBaumZeiger; wert: Integer): Boolean;
begin
  if knoten = nil then
    Result := False
  else if knoten^.inhalt = wert then
    Result := True
  else
    Result := Enthaelt(knoten^.links, wert) or Enthaelt(knoten^.rechts, wert);
end;
```

Im schlechtesten Fall werden alle n Knoten besucht. Abschnitt 3.4 zeigt, wie viel billiger das wird, sobald der Baum geordnet ist.

Merke

Alle Algorithmen auf Bäumen folgen demselben Muster:

```
wenn der Baum leer ist:  trivialer Fall
sonst:                   etwas mit dem Knoten tun,
                         dasselbe für links,
                         dasselbe für rechts
```

Was Sie verändern, ist nur, *was* Sie mit dem Knoten tun und *wann*.

3.3.2 Die drei Durchläufe

Das „wann" aus dem Merksatz ist bereits die ganze Theorie der Traversierungen. Man kann den Knoten vor, zwischen oder nach seinen Teilbäumen besuchen — daraus ergeben sich drei Reihenfolgen.

Definition: Preorder, Inorder, Postorder

- **Preorder**: Wurzel — linker Teilbaum — rechter Teilbaum
- **Inorder**: linker Teilbaum — Wurzel — rechter Teilbaum
- **Postorder**: linker Teilbaum — rechter Teilbaum — Wurzel

```python
def preorder(knoten):
    if knoten is not None:
        print(knoten.inhalt)            # Wurzel zuerst
        preorder(knoten.links)
        preorder(knoten.rechts)

def inorder(knoten):
    if knoten is not None:
        inorder(knoten.links)
        print(knoten.inhalt)            # Wurzel in der Mitte
        inorder(knoten.rechts)

def postorder(knoten):
    if knoten is not None:
        postorder(knoten.links)
        postorder(knoten.rechts)
        print(knoten.inhalt)            # Wurzel zuletzt
```

Drei Methoden, die sich in **einer einzigen Zeile** unterscheiden: in der Position des `print`.

Beispiel: Die drei Durchläufe am Baum von oben

Durchlauf	Reihenfolge
preorder	A B D E C F G
inorder	D B E A G F C
postorder	D E B G F C A

Probe für postorder: Die Wurzel A steht ganz hinten — so muss es sein, sie wird zuletzt besucht. Für preorder steht sie ganz vorn.

Tragen Sie die Inorder-Reihenfolge des Baums ein und prüfen Sie: Steht A in der Mitte?

Arbeitsauftrag 4

10. Zeichnen Sie den Baum, geben Sie alle drei Durchläufe an und vergleichen Sie mit der Tabelle.
11. Schreiben Sie eine Anfrage `blaetter(knoten)`, die die Zahl der Blätter liefert.
12. Schreiben Sie eine Methode, die einen Baum spiegelt (links und rechts vertauschen). Welcher der drei Durchläufe verändert sich dabei wie?

3.4 Suchbäume

Bisher stand der Inhalt beliebig im Baum. Ordnet man ihn, entsteht eine Struktur, in der man sehr schnell suchen kann.

Definition: Binärer Suchbaum

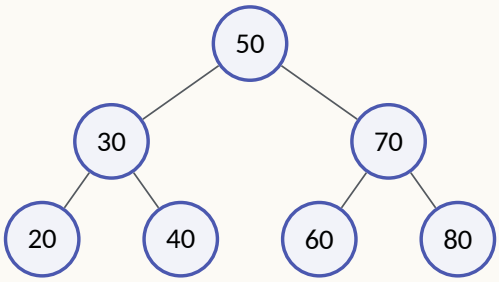
Ein binärer Baum heißt **Suchbaum**, wenn für **jeden** Knoten gilt:

- alle Inhalte im **linken** Teilbaum sind **kleiner** als der Inhalt des Knotens,
- alle Inhalte im **rechten** Teilbaum sind **größer**.

Die Bedingung muss für jeden Knoten gelten, nicht nur für die Wurzel — und für den ganzen Teilbaum, nicht nur für die direkten Nachfolger.

Beispiel: Aufbau durch Einfügen

Fügt man nacheinander 50, 30, 70, 20, 40, 60, 80 ein, entsteht:



Links steht immer das Kleinere, rechts das Größere — und zwar im ganzen Teilbaum.

Suchbaum aus 50, 30, 70, 20, 40, 60, 80 — in dieser Reihenfolge eingefügt.

Gymnasium Ottweiler

Merke

Der **Inorder**-Durchlauf eines Suchbaums liefert die Inhalte in **aufsteigender Reihenfolge**. Für den Baum oben: 20, 30, 40, 50, 60, 70, 80.

Das ist keine Zufälligkeit, sondern die Definition: links steht das Kleinere, also wird es zuerst besucht. Ein Suchbaum sortiert, indem man ihn richtig ausliest.

> In der Webfassung steht hier ein Bedienfeld zum Ausprobieren: **Suchbaum-Werkzeug**.

3.4.1 Suchen

```python
def suchen(knoten, wert):
    if knoten is None:
        return False                    # nicht gefunden
    if wert == knoten.inhalt:
        return True
    if wert < knoten.inhalt:
        return suchen(knoten.links, wert)
    return suchen(knoten.rechts, wert)
```

```
function Suchen(knoten: TBaumZeiger; wert: Integer): Boolean;
begin
  if knoten = nil then
    Result := False
  else if wert = knoten^.inhalt then
    Result := True
  else if wert < knoten^.inhalt then
    Result := Suchen(knoten^.links, wert)
  else
    Result := Suchen(knoten^.rechts, wert);
end;
```

Beispiel: Die 60 suchen

```
50 → 60 ist größer → nach rechts
70 → 60 ist kleiner → nach links
60 → gefunden
```

Drei Vergleiche bei sieben Einträgen. Jeder Vergleich halbiert, was noch zu durchsuchen bleibt — genau wie bei der binären Suche in Abschnitt 4.1.2.

3.4.2 Einfügen

Eingefügt wird immer als neues Blatt. Man sucht die Stelle, an der der Wert *stünde*, wenn er da wäre — und hängt ihn dort an.

```
def einfuegen(knoten, wert):
    if knoten is None:
        return BaumKnoten(wert)          # hier ist der Platz
    if wert < knoten.inhalt:
        knoten.links = einfuegen(knoten.links, wert)
    elif wert > knoten.inhalt:
        knoten.rechts = einfuegen(knoten.rechts, wert)
    return knoten                        # wert == inhalt: nichts tun
```

Stolperstein

Die Reihenfolge beim Einfügen bestimmt die Form des Baums. Fügt man 20, 30, 40, 50, 60, 70, 80 **der Größe nach** ein, hängt jeder Knoten rechts am vorigen — der Baum entartet zur Liste, und die Suche braucht statt drei Vergleichen bis zu sieben.

Der Suchbaum ist also nur so gut wie seine Form. Es gibt Verfahren, die ihn beim Einfügen selbsttätig wieder ausgleichen — AVL-Bäume und Rot-Schwarz-Bäume. Sie stehen in keinem der beiden Lehrpläne, geben aber ein lohnendes Thema für eine Facharbeit ab.

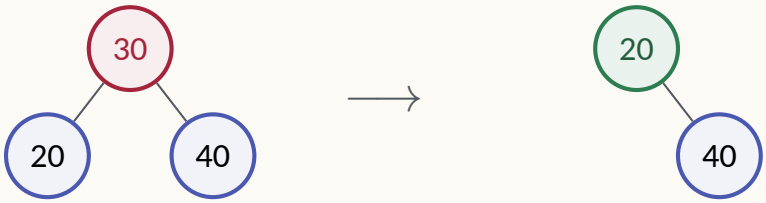
3.4.3 Löschen

Das Löschen ist der einzige unangenehme Fall, weil drei Situationen zu unterscheiden sind:

Fall	Vorgehen
Der Knoten ist ein **Blatt**	einfach entfernen
Der Knoten hat **einen** Nachfolger	dieser rückt samt seinem ganzen Teilbaum nach
Der Knoten hat **zwei** Nachfolger	siehe unten

Im dritten Fall darf der Knoten nicht einfach verschwinden — es muss jemand nachrücken, der die Suchbaumbedingung nicht verletzt. Dafür kommen genau zwei Knoten in Frage: der **größte Knoten des linken Teilbaums** oder der **kleinste des rechten**. Wir nehmen den ersten. Sein Inhalt wandert in den zu löschenden Knoten, und er selbst wird an seiner alten Stelle gelöscht — was leicht ist, denn er hat höchstens einen Nachfolger.

Beispiel: Die 30 löschen



vorher nachher

Die 30 hat zwei Nachfolger. Der größte Knoten ihres linken Teilbaums ist die 20 — sie rückt nach, ihr alter Platz wird frei. Probe über den Inorder-Durchlauf: vorher 20, 30, 40, nachher 20, 40. Aufsteigend, die Bedingung hält.

Löschen eines Knotens mit zwei Nachfolgern.

Merke

Prüfen Sie jede Veränderung an einem Suchbaum mit dem Inorder-Durchlauf. Kommt dabei keine aufsteigende Folge heraus, ist der Baum kaputt.

Arbeitsauftrag 5

13. Bauen Sie einen Suchbaum aus 55, 20, 80, 10, 35, 70, 95, 30, 40 auf. Zeichnen Sie ihn und geben Sie Tiefe und Knotenzahl an.
14. Geben Sie zu Ihrem Baum alle drei Durchläufe an. Welcher ist aufsteigend sortiert?
15. Löschen Sie daraus die 20. Zeichnen Sie das Ergebnis und prüfen Sie es mit dem Inorder-Durchlauf.
16. Implementieren Sie `einfuegen` und `suchen` in einer Sprache Ihrer Wahl und lesen Sie den Baum inorder aus. Sie haben damit nebenbei ein Sortierverfahren geschrieben.
17. In welcher Reihenfolge müssten Sie 1 bis 7 einfügen, damit ein möglichst *flacher* Baum entsteht? Welche Tiefe erreichen Sie?

Selbstcheck

Können Sie ohne nachzusehen angeben, welcher Durchlauf einen Suchbaum sortiert ausliest — und begründen, warum das aus der Definition folgt? Und können Sie erklären, weshalb die Rekursion auf Bäumen so viel natürlicher wirkt als auf Feldern?