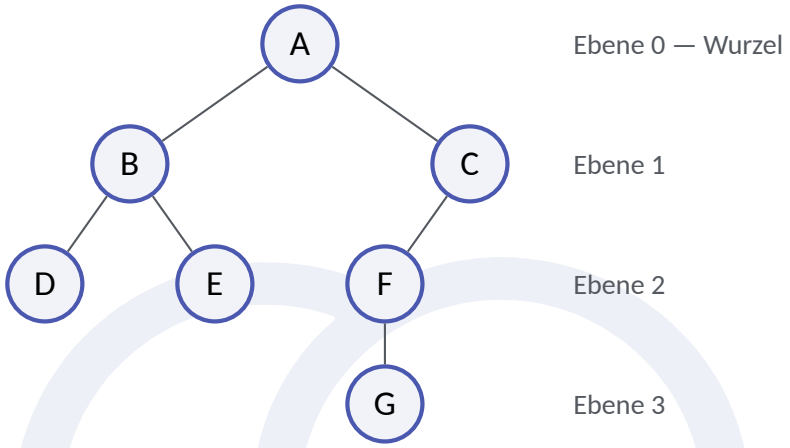
3.3 Binäre Bäume

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?