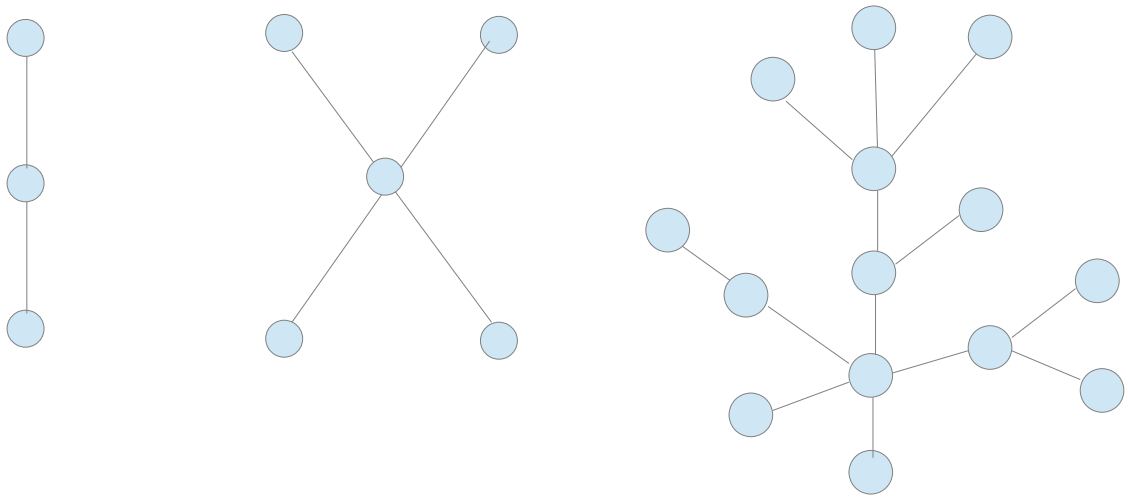
Lektion 5

In dieser Lektion wird das Konzept der Bäume betrachtet, einer besonderen Art von Graphen, die an vielen Stellen in der Informatik verwendet werden. Eine spezielle Form von Bäumen sind *Binärbäume*, die z.B. bei effizienten Datenstrukturen eine wichtige Rolle spielen.

3.6 Bäume

Bäume sind eine spezielle Art von "verzweigten" Graphen, die in der Informatik an vielen Stellen verwendet werden.

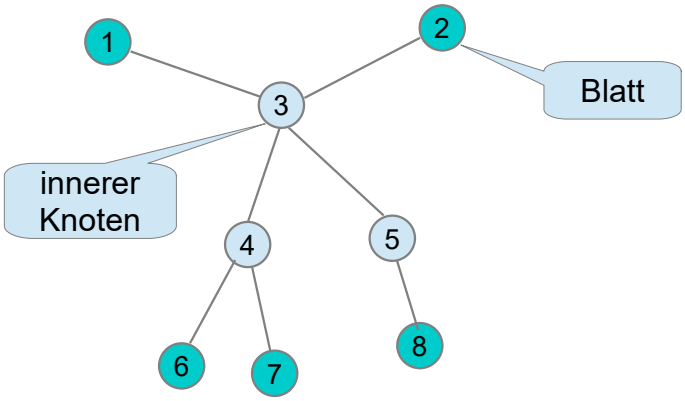
Beispiel 3.38 - Bäume



Definition 3.39 - Bäume, Blätter und innere Knoten

- Ein **Baum** ist ein ungerichteter, zusammenhängender Graph ohne Schlingen, der **keine nichttrivialen Kreise** enthält.
  (Ein Kreis heißt *nichttrivial*, wenn er mehr als zwei Knoten verbindet.)
- Alle Knoten mit Grad 1 nennt man **Blätter** des Baums, die anderen Knoten heißen **innere Knoten**.

Beispiel 3.40 - Baum



Anmerkung

Diese sehr abstrakte und wenig anschauliche Definition bedeutet folgendes.

- Hat man zwei verschiedene Knoten eines Baum, dann gibt es genau eine Möglichkeit, vom einem Knoten zum anderen zu kommen.
- Nehmen wir beispielsweise Knoten 4 und 8 in Beispiel 3.40. Die einzige Möglichkeit, um von 4 nach 8 zu kommen, ist der Pfad (4,3,5,8). Sobald man ein weitere Kante zwischen zwei Knoten dazu nehmen würde, hätte man in manchen Fällen auch mehrere unterschiedliche Wege zwischen zwei Knoten.

Für alle Bäume gilt der folgende, sehr einfache Zusammenhang zwischen der Kanten- und Knotenzahl.

Satz 3.41 - Anzahl Knoten und Kanten eines Baums

Jeder nicht leere Baum mit ***n* Knoten** hat ***n*-1 Kanten**.

Beweis

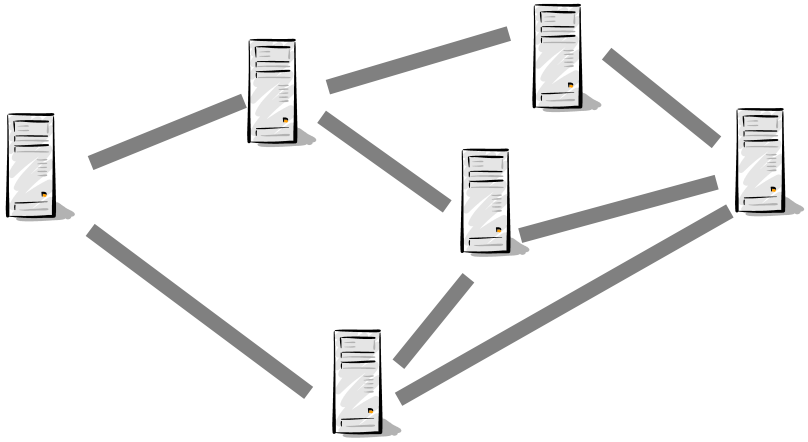
Diese Eigenschaft lässt sich leicht mit vollständiger Induktion über die Anzahl der Knoten beweisen.

- **Fall *n* = 1 (Induktionsbasis)**: Bei nur einem Knoten kann der Baum keine Kante haben, da eine Schlinge als Kreis ausgeschlossen ist.
- **Fall n > 1 (Induktionsschritt)**: Der Baum hat mindestens ein Blatt. Entfernt man ein Blatt und die dazugehörige Kante, erhält man wieder einen Baum, der dann *n*-1 Knoten und einer Kante weniger hat. Nun können wir als Induktionshypothese verwenden, dass die Eigenschaft für *n*-1 Knoten gilt, d.h. der verbleibende Baum hat *n*-2 Kanten. Folglich hatte der ursprüngliche Baum (*n*-2) + 1 = *n* - 1 Kanten.

Somit folgt, dass die Eigenschaft für alle Bäume mit 1 oder mehr Knoten gilt. □

Motivation - Kommunikationsnetz

Kommunikationsnetze mit Rechnerknoten und direkten Kommunikationsverbindungen dazwischen, wie beispielsweise das Internet, können als ungerichtete Graphen modelliert werden. Nachrichten werden dabei ggf. über mehrere Rechnerknoten vom Sender an den Empfänger weitergeleitet.



Ein wichtiger Aspekt dabei ist die Frage nach der Robustheit des Netzwerks gegen den Ausfall einzelner Verbindungen. Fragestellungen in dem Zusammenhang sind z.B.:

- Welche direkten Verbindungen dürfen ausfallen, so dass die Verbindung zwischen allen Rechnern noch aufrecht erhalten werden kann?
- Wie viele Einzelverbindungen braucht man mindestens, damit noch jeder Rechner mit jedem anderen kommunizieren kann?

Damit jeder Knoten noch mit jedem kommunizieren kann, muss es ein zusammenhängender Graph bleiben. Die minimale Anzahl an einzelnen Direktverbindungen (= Kanten in der Darstellung als Graph) hat man dann, wenn das Kommunikationsnetz ein Baum ist. Würde man dann eine weitere Verbindung entfernen, wäre es nicht mehr zusammenhängend, würde man eine zusätzliche Verbindung dazunehmen, wäre es nicht mehr die minimal Anzahl, die die Verbindungsfähigkeit gewährleistet. Mathematisch führt das zum Konzept der Spannbäume.

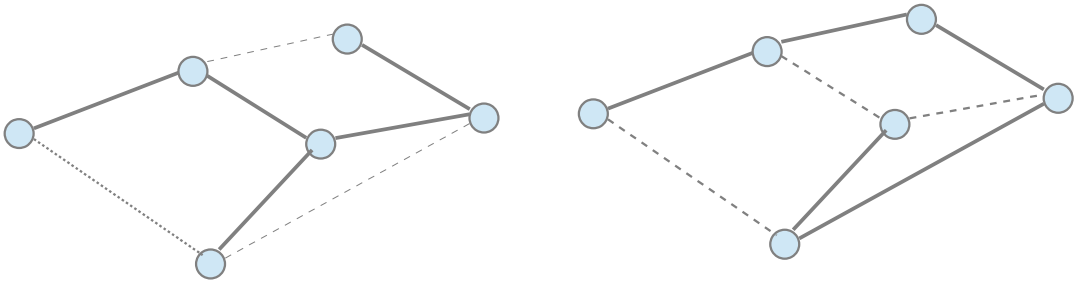
Definition 3.42 - Spannbäume

Ein **Spannbaum** eines ungerichteten, zusammenhängenden Graphen $G = (V, E)$ ist ein Teilgraph $G' = (V', E')$ mit $\boldsymbol{V' = V}$, der ein **Baum** ist.

Spannbaum für einen Graph G heißt also, man behält alle Knoten bei und nimmt von den Kanten so viele weg, dass ein Baum übrig bleibt.

Beispiel 3.43 - Spannbäume

Hier sind zwei Spannbäume für das oben angegebene Kommunikationsnetz angegeben:



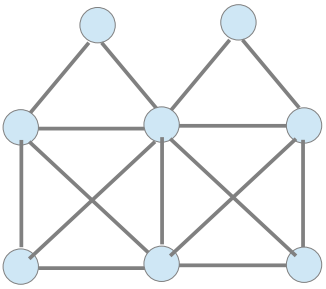
Anmerkung

Besteht ein Kommunikationsnetz aus *n* Knoten, so braucht man somit mindestens *n*-1 direkte Verbindungen, um zu gewährleisten, dass jeder mit jedem kommunizieren kann, denn jede minimale Verbindungsstruktur ist ein Spannbaum, und jeder Spannbaum mit *n* Knoten hat *n*-1 Kanten.

Je nachdem, welche Verbindungen ausfallen, kann es natürlich auch Situationen geben, dass zwar noch *n*-1 oder mehr Verbindungen bestehen, trotzdem nicht alle miteinander verbunden sind.

Aufgabe 3.44 - Spannbäume

(1) Wie viele Kanten und Knoten hat ein Spannbaum für das Doppelhaus vom Nikolaus?



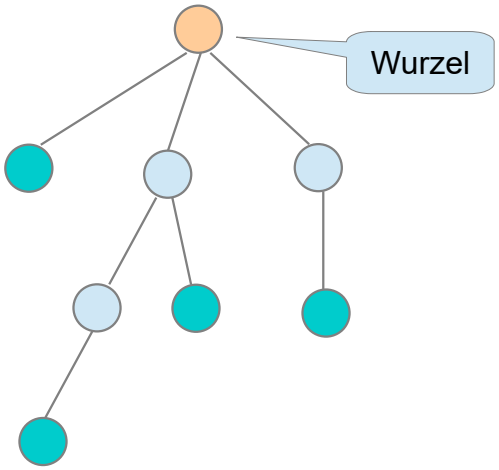
(2) Geben Sie einen Spannbaum dafür an.

3.7 Wurzelbäume

Um streng hierarchisch aufgebaut Strukturen zu beschreiben (und das kommt in der Informatik häufig vor, denken Sie an das Dateisystem mit Verzeichnissen und Unterverzeichnissen), werden **gewurzelte Bäume** verwendet.

- Ein gewurzelter Baum hat einen ausgezeichneten Knoten als **Wurzel**.
- Jeder Knoten kann beliebig viele Nachfolger haben. Die Nachfolger eines Knotens nennt man auch seine **Kinder** (bzw. Söhne/Töchter).

- Jeder Knoten, außer der Wurzel, hat genau einen Knoten als **Elternknoten** (auch Vater-/Mutterknoten, Vorgänger).
- **Blätter** sind also Knoten ohne Kinder, **innere Knoten** sind die Knoten, die Kinder haben.
- Bäume werden üblicherweise so dargestellt, dass die Wurzel oben ist und die Blätter unten sind.
- In der Informatik sind mit "Baum" meist gewurzelte Bäume gemeint.



Durch die Wurzel ergibt sich implizit eine Richtung "von der Wurzel weg". Die formale Definition basiert deshalb auf gerichteten Graphen.

Definition 3.45 - gewurzelter Baum/Wurzelbaum

Ein gerichteter, zusammenhängender, azyklischer Graph $G = (V, E)$ ist ein **Wurzelbaum**, wenn

- es genau einen Knoten w mit Eingangsgrad 0 (die **Wurzel**) gibt, und
- alle anderen Knoten den Eingangsgrad 1 haben.

Darstellung von gewurzelten Bäumen

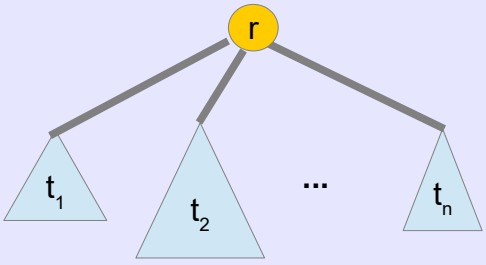
In der zeichnerischen Darstellung werden die Kanten üblicherweise nicht als Pfeile gezeichnet, die Richtung ist durch die Darstellung "von oben nach unten" angegeben.

3.7.1 Bäume als rekursive Datenstruktur

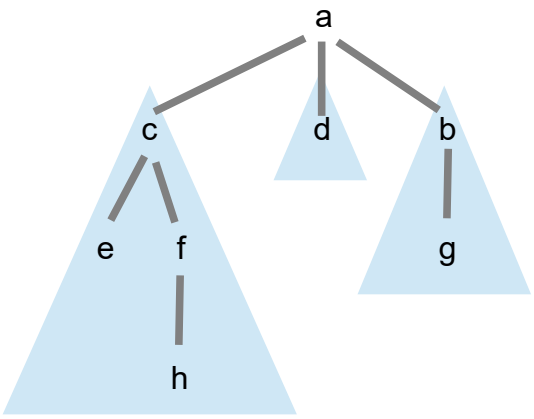
Gewurzelte Bäume kann man auch als eine rekursiv definierte Datenstruktur betrachten. Rekursion gibt es nicht nur bei Methoden, sondern auch bei Datenstrukturen.

Definition 3.46 - Rekursive Definition für gewurzelte Bäume

- Der leerer Graph (mit leerer Knoten- und Kantenmenge) ist ein Baum (leerer Baum)
- Sind t_1 , t_2, … t_n Bäume (für $n \geq 0$) und ist r ein neuer Knoten, dann ist folgendes auch ein Baum mit Wurzel r, wobei die Kanten jeweils von r zu den Wurzeln der Teilbäume führen.



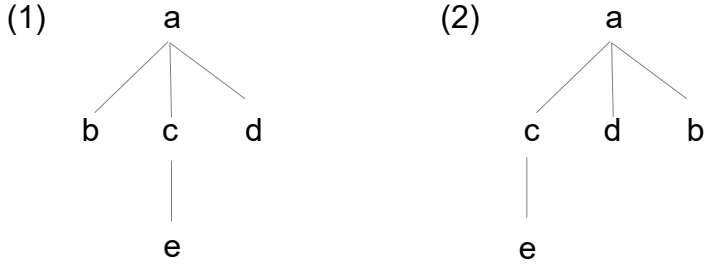
Beispiel 3.47 - Baum mit Teilbäumen



Passt diese rekursive Definition auch, wenn man Blätter betrachtet, d.h. (Teil)bäume, die nur aus einem Knoten bestehen? Ja - in diesem Fall ist es so zu verstehen, dass die Menge der Teilbäume leer ist.

3.7.2 Geordnete Bäume

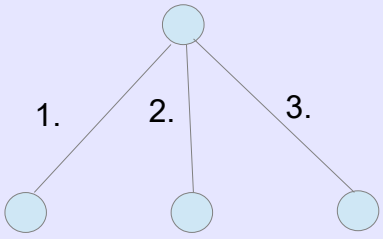
Geht man von der mathematischen Definition von Graphen aus, dann wären durch folgende Darstellungen zweimal der gleiche gewurzelte Baum angegeben, da jeweils gleich ist, wie die Knoten durch Kanten verbunden sind.



Es gibt Arten von Bäumen, bei denen die Reihenfolge der Kinder Bedeutung hat.

Definition 3.48 - Geordnete Bäume

Bei **geordneten Bäumen** sind die Kinder eines Knotens in einer Reihenfolge geordnet, d.h. man unterscheidet erstes, zweites, …, *n*-tes Kind.



Die Reihenfolge wird in der grafischen Darstellung durch die Position von links nach rechts angegeben.

Als *geordnete* Bäume wären (1) und (2) oben also unterschiedliche Bäume. Bei (1) ist Knoten *b* das erste Kind von *a*, bei (2) ist Knoten *b* das dritte Kind.

3.7.3 Tiefe von Knoten und Höhe von Bäumen

Bei Bäumen mit Wurzel ergeben sich naheliegend zwei Begriffe: *Tiefe* eines Knoten und *Höhe* des Baum.

Definition 3.49 - Tiefe eines Knotens im Baum

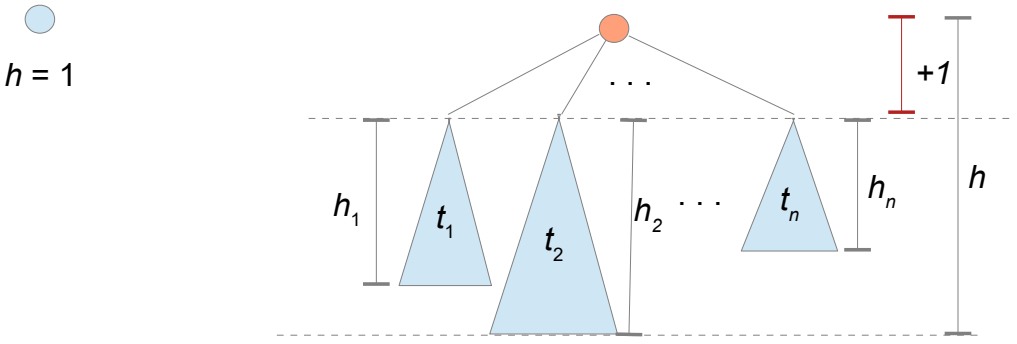
Die **Tiefe** eines Knotens *k* in einem Baum ist die Länge des Pfads von der Wurzel zum Knoten *k*.

- Die *Tiefe* eines Knotens gibt also den Abstand zur Wurzel an. Die Wurzel hat somit Tiefe 0.

Definition 3.50 - Höhe von Bäumen

Die **Höhe** *h* eines Wurzelbaums ist folgendermaßen rekursiv definiert:

- Ein leerer Baum (ohne Knoten) hat die Höhe $h = 0$.
- Ein Baum aus einem einzigen Knoten (der Wurzel) hat Höhe $h = 1$.
- Besteht der Baum aus der Wurzel *w* und den Teilbäumen $t_1, ..., t_n$ und hat Teilbaum t_i die Höhe h_i, dann hat der Baum die Höhe
  $h = 1 + \max\{h_1, ..., h_n\}$.

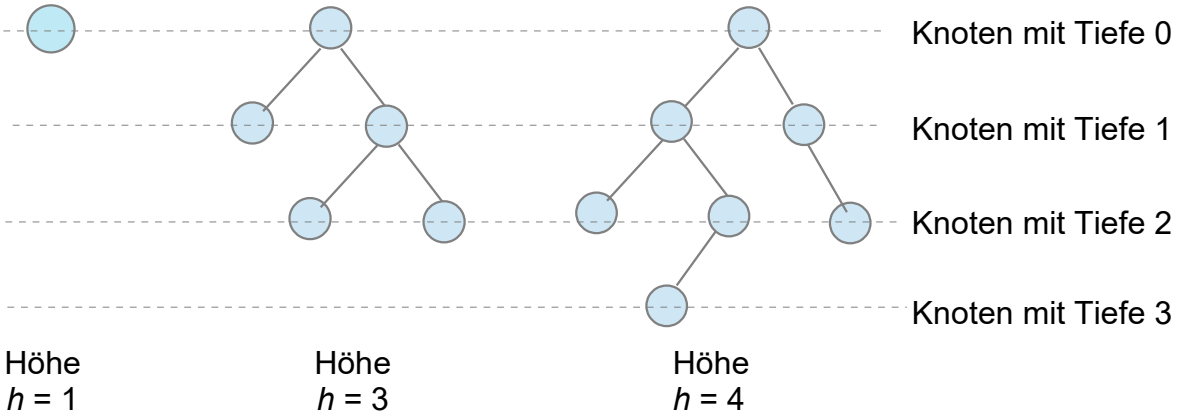


$h = 1 + \max(h_1, ..., h_n)$.

Anmerkung - Zusammenhang Höhe und Tiefe

Die Höhe eines Baums könnte auch so definiert werden:

Höhe des Baums = 1 + maximale Tiefe aller Knoten des Baums

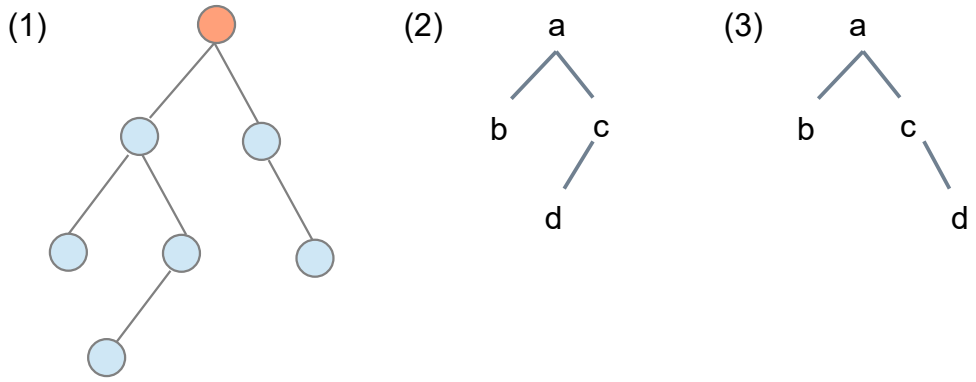


3.8 Binärbäume

Definition 3.51 - Binärbäume

Ein **Binärbaum** ist ein geordneter Wurzelbaum, dessen Knoten einen Ausgangsgrad von höchstens 2 haben, d.h. ein Knoten kann maximal zwei Kinder haben.

Beispiel 3.52 - Binärbaum



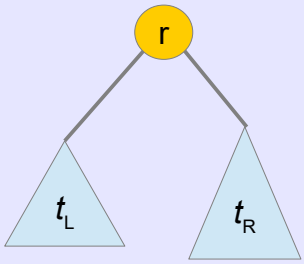
Anmerkungen

- Binärbäumen sind geordnet, d.h. es wird zwischen dem linken und dem rechten Nachfolger eines Knotens unterschieden. Beispielweise werden die Bäume (2) und (3) oben unterschieden. Bei (2) ist *d* linkes Kind von *c*, bei (3) rechtes Kind.

Die rekursive Definition für Bäume lässt sich auch auf Binärbäume übertragen.

Definition 3.53 - Binärbäume rekursiv definiert

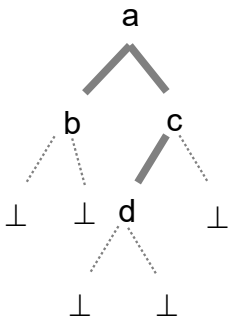
- Der leere Baum ist ein Binärbaum.
- Sind t_L und t_R Binärbäume und ist r ein neuer Knoten, dann ist auch folgendes ein Binärbaum (mit Wurzel r) mit linkem Teilbaum t_L und rechtem Teilbaum t_R:



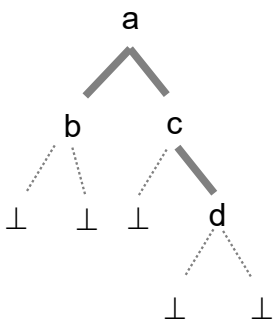
Beispiel 3.54 - Binärbäume rekursiv

Passt diese Definition zu den oben angegebenen Beispielen für Binärbäume? Ja, die beiden Bäume müssen dazu folgendermaßen gesehen werden: Hat ein Knoten links oder rechts keinen Nachfolger, ist der entsprechende Teilbaum der leere Baum.

(1)



(2)



⊥ leerer Baum

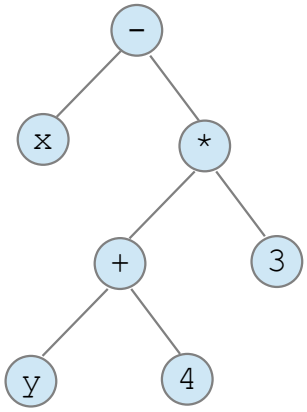
In Programmieren 2 werden Sie binäre Bäume implementierten. Dort wird diese Sichtweise auch verwendet (leerer Baum entspricht dort einer null-Referenz).

Abstrakte Syntaxbäume

Eine Anwendung von Bäumen im Umfeld der Programmiersprachen-Implementierung sind sog. *abstrakte Syntaxbäume* (engl. *abstract syntax tree*, *AST*), die dazu verwendet werden, den strukturellen Aufbau von Programmteilen zu beschreiben. Betrachtet man arithmetische oder logische Ausdrücke mit zweistelligen Operatoren, ergeben sich binäre Bäume.

Beispiel 3.55 - Abstrakte Syntaxbäume

- Arithmetischer Ausdruck `x - (y + 4) * 3`:



- Dieser Baum repräsentiert die richtige Leseweise entsprechend der üblichen Regel "Punkt-vor-Strichrechnung".
- Die Klammern in der Textdarstellung dienen nur dazu, die richtige Strukturierung anzugeben, im abstrakten Syntaxbaum tauchen sie deshalb nicht auf.

Im Allgemeinen kann es in abstrakten Syntaxbäumen beispielsweise auch einstellige Operationen geben oder Sprachkonstrukten, die mehr als zwei Bestandteile haben. Das wird dann analog behandelt. Der Abstrakte Syntaxbaum ist dann zwar noch ein (gewurzelter) Baum, aber kein binärer Baum mehr.

Aufgabe 3.56 - Abstrakter Syntaxbaum

Zeichnen Sie einen abstrakten Syntaxbaum für den mathematischen Ausdruck

$$2x^3 + \sqrt{y+7} \quad .$$

3.8.1 Vollständige Binärbume

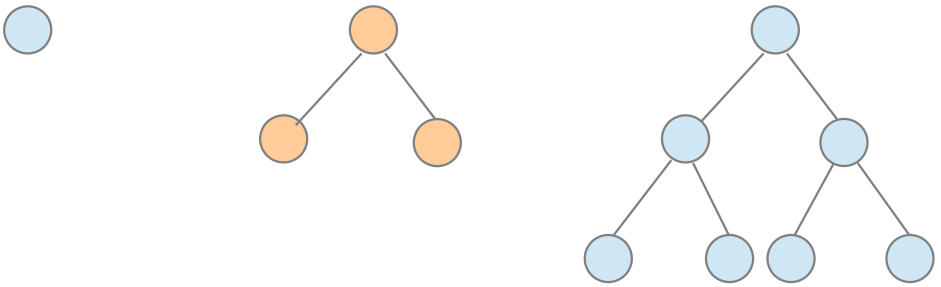
Auch bei Binärbäumen gibt es den Begriff der Vollständigkeit. Ein Binärbaum ist vollständig, wenn er bis zu einer "Ebene" vollständig mit Knoten gefüllt ist.

Definition 3.57 - Vollständiger Binärbaum

Ein ***Binärbaum*** heißt **vollständig**, wenn

- ❒ alle innere Knoten den Ausgangsgrad 2 haben und
- ❒ alle Blätter die gleiche Tiefe haben (gleichen Abstand zur Wurzel).

Beispiel 3.58 - vollständige Binärbaume



vollständiger Binärbaum der Höhe h = 1

vollständiger Binärbaum der Höhe h = 2

vollständiger Binärbaum der Höhe h = 3

Offensichtlich verdoppelt sich bei vollständigen Binärbäumen jeweils die Zahl der Blätter, wenn die Höhe um 1 zunimmt. Daraus ereben sich folgende Eigenschaften.

Satz 3.59 Eigenschaften vollständiger Binärbäume

Ein **vollständiger Binärbaum** der Höhe $h > 0$ hat

- $\mathbf{2^{(h-1)}}$ **Blätter** und
- $\mathbf{2^h - 1}$ **Knoten**.

Beweis

Vollständige Induktion über die Höhe *h* des vollständigen Binärbaums:

***Fall h* = 1**: Baum hat nur einen Knoten, der auch Blatt ist.

$1 = 2^0 = 2^{h-1}$ Blätter
$1 = 2^1 - 1 = 2^h - 1$ Knoten.

***Fall h* = *n*+1**: Baum besteht aus Wurzel und zwei Teilbäumen der Höhe *n*.

Jeder Teilbaum ist ein vollständiger Binärbaum der Höhe *n* und hat nach Induktionshypothese somit $2^{(n-1)}$ Blätter und $2^n - 1$ Knoten.

Für den gesamten Baum ergibt sich:

Anzahl Blätter = Blätter linker Teilbaum + Blätter rechter Teilbaum:

$$\begin{aligned} & 2^{n-1} + 2^{n-1} \\ = \; & 2 \cdot 2^{n-1} = 2^n \\ = \; & 2^{h-1} \end{aligned}$$

Anzahl Knoten: Wurzel + Knoten linker Teilbaum + Knoten rechter Teilbaum

$$\begin{aligned} & 1 + (2^n - 1) + (2^n - 1) \\ = \; & 1 + 2 \cdot 2^n - 2 = 2^{n+1} - 1 \\ = \; & 2^h - 1 \end{aligned}$$

□

3.8.2 Traversierung von Binärbäumen

Bei der Verarbeitung von Binärbäumen muss oft der komplette Baum durchlaufen und jeder Knoten irgendwie verarbeitet werden. Orientiert an der rekursiven Definition von Bäumen (nichtleerer Baum besteht aus Wurzel, linkem und rechtem Teilbaum), ergeben sich folgende Standard-Reihenfolgen, um Binärbäume rekursiv komplett zu durchlaufen und alle Knoten zu verarbeiten.

Definition 3.60 - Rekursive Verarbeitung von Binärbäumen

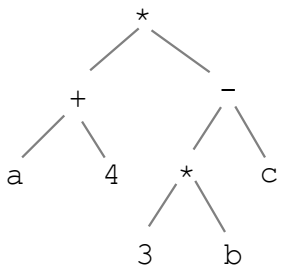
- **Preorder-Traversierung:** falls Baum nicht leer,
  (1) verarbeite Wurzel
  (2) durchlaufe linken Teilbaum in Preorder (rekursiv),
  (3) durchlaufe rechten Teilbaum in Preorder (rekursiv)
- **Inorder-Traversierung**: falls Baum nicht leer,
  (1) durchlaufe linken Teilbaum in Inorder-Reihenfolge (rekursiv)
  (2) verarbeite Wurzel
  (3) durchlaufe rechten Teilbaum in Inorder-Reihenfolge (rekursiv)
- **Postorder-Traversierung:** falls Baum nicht leer,
  (1) durchlaufe linken Teilbaum in Postorder (rekursiv)
  (2) durchlaufe rechten Teilbaum in Postorder (rekursiv)
  (3) verarbeite Wurzel

Allen drei Traversierungsreihenfolgen ist gemeinsam, dass bei einem nicht leeren (Teil-)Baum, der eine Wurzel *w* und zwei Teilbäume L und R hat, sowohl die Wurzel als auch rekursiv die Knoten in den beiden Teilbäumen bearbeitet werden (Teilbäume immer erst links, dann rechts). Der Unterschied liegt darin, wann die Wurzel d'rankommt:

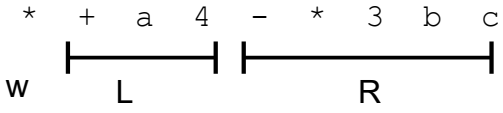
Preoder (pre = vor):	w – L – R
Inorder (in = dazwischen):	L – w – R
Postorder (post = danach):	L – R – w

Beispiel 3.61 - Arithmetische Ausdrücke

- abstrakter Syntaxbaum für Ausdruck `(a+4)*(3*b-c)`:



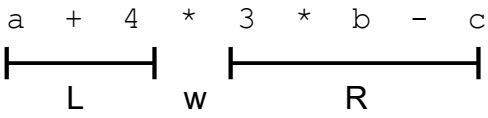
- Durchlauf in *Preorder*-Reihenfolge:



Die Preorder-Traversierung liefert ein Ergebnis ähnlich der funktionalen Notation in der Mathematik oder in funktionalen Programmiersprachen (z.B. LISP).

```
*( +(a,4), -(*(3,b), c))
```

- Durchlauf in *Inorder*-Reihenfolge:

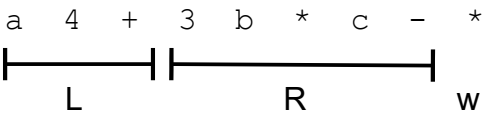


Von Klammern abgesehen, ergibt das die übliche Infix-Notation:

```
((a + 4) * ((3 * b) - c))
```

(hier mit Klammern um jeden Teilbaum)

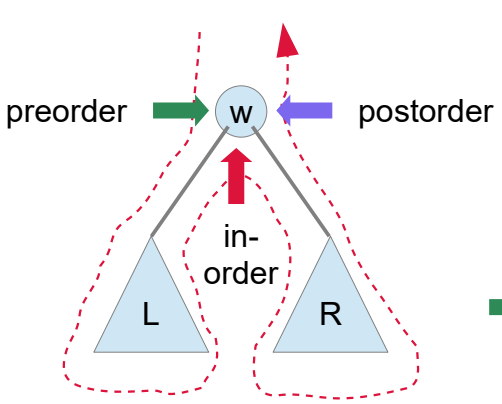
- Durchlauf in *Postorder*-Reihenfolge:

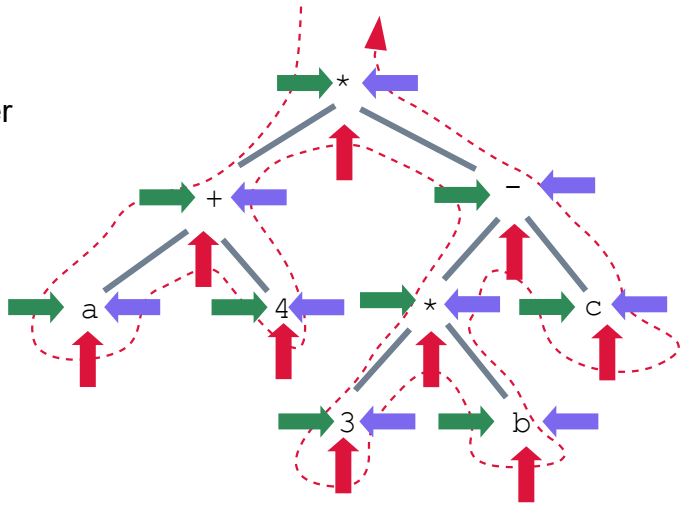


Postorder ergibt die sog. *Umgekehrt Polnische Notation* (UPN), die im Compilerbau bei der Codegenerierung eine wichtige Rolle spielt. Soll ein Teilausdruck mit einem Operator und zwei Operanden ausgewertet werden, dann müssen erst die beiden Operanden berechnet werden, bevor die Operation ausgeführt werden kann (siehe Exkurs Codegenerierung unten).

Veranschaulichung der Traviersierungsreihenfolgen

Bei allen drei Traversierungsfolgen wird der Baum im Wesentlichen gleichartig rekursiv durchlaufen. Die Unterschied liegen nur darin, wann die Wurzel eines (Teil-)Baums behandelt wird: ob vor, zwischen oder nach den beiden Teilbäumen.





3.8.3 Exkurs: Codegenerierung für Stack-Maschinen

Wie Sie aus Programmieren 1 in Erinnerung haben, wir bei Java der Quelltext zunächst in den sog. *Bytecode* übersetzt und dieser Code dann von der *Java-VM* (virtual machine), einer abstrakten Maschine, ausgeführt.

Die Grundarchitektur der Java-VM ist eine **Stack-Machine**. Das bedeutet, ein sog. *Stack* wird bei der Auswertung von Ausdrücken zur Zwischenspeicherung von Teilergebnissen verwendet und Operationen verwenden als Operanden immer die obersten Einträgen des Stacks.

Ein Stack ist eine Datenstruktur für eine Sammlung von Daten, die nach dem *LIFO*-Prinzip ("Last in, first out") verwaltet wird:

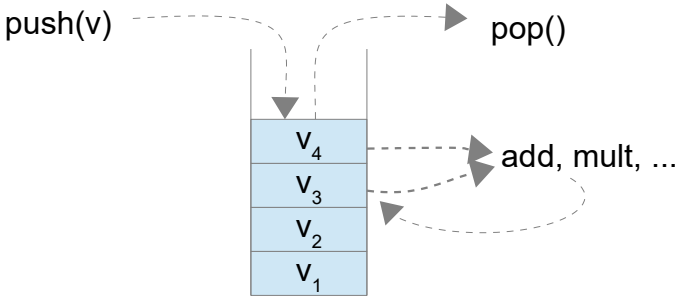
- Mit *push*(x) wir ein Wert x oben auf den Stack gelegt,
- mit *pop*() wird das oberste Element entnommen.

(Vergleichbar zu einem Stapel von Tellern, wenn man entweder einen Teller oben auf den Stapel legen kann oder den obersten Teller des Stapels entfernen kann.)

Die Vorteile einer Stack-Maschine sind, dass ein Stack sehr einfach und effizient zu realisieren ist und die Maschinenbefehle sehr einfach gehalten werden können, da die Operanden von Operationen nicht explizit angegeben werden müssen, so dass pro Befehl in den meisten Fällen nur ein Byte nötig ist (daher die Bezeichnung *Bytecode*). Etwas vereinfacht – aber gar nicht so weit von der Java-Realität entfernt – sieht das so aus:

Stack-Machine

- Stack speichert Zwischenergebnisse für die Auswertung



- Operationen:

`const` *`n`*	legt Zahl *n* oben auf dem Stack ab
`load` `var`	legt Inhalt von Variable *var* oben auf den Stack
`add`	entnimmt die zwei obersten Werte vom Stack, bildet die Summe und legt Ergebnis auf den Stack
`sub`	subtrahiert entsprechend die zwei obersten Werte des Stacks
`mult`	multipliziert die zwei obersten Werte

...

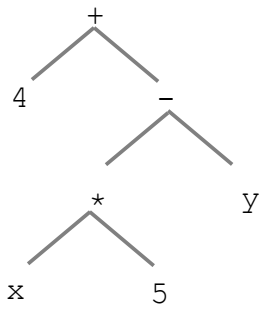
- Die **Codegenerierung** für Ausdrücke erfolgt durch eine **Postorder-Traversierung** des abstrakten Syntaxbaums. Je nach Art des Knotens wird dabei ein entsprechender Befehl generiert:

Knoten	zu generierender Code
Zahl *n*	**`const`** *n*
Variable *var*	**`load`** *var*
Operation +	**`add`**
Operation -	**`sub`**
Operation `*`	**`mult`**
...	...

Beispiel 3.62 - Bytecode-Generierung und Ausführung

Für Ausdruck `4 + (x * 5 - y)` soll Bytecode s.o. generiert werden und dann der Code ausgeführt werden:

- Codegenerierung: Ausgangspunkt ist der abstrakte Syntaxbaum:



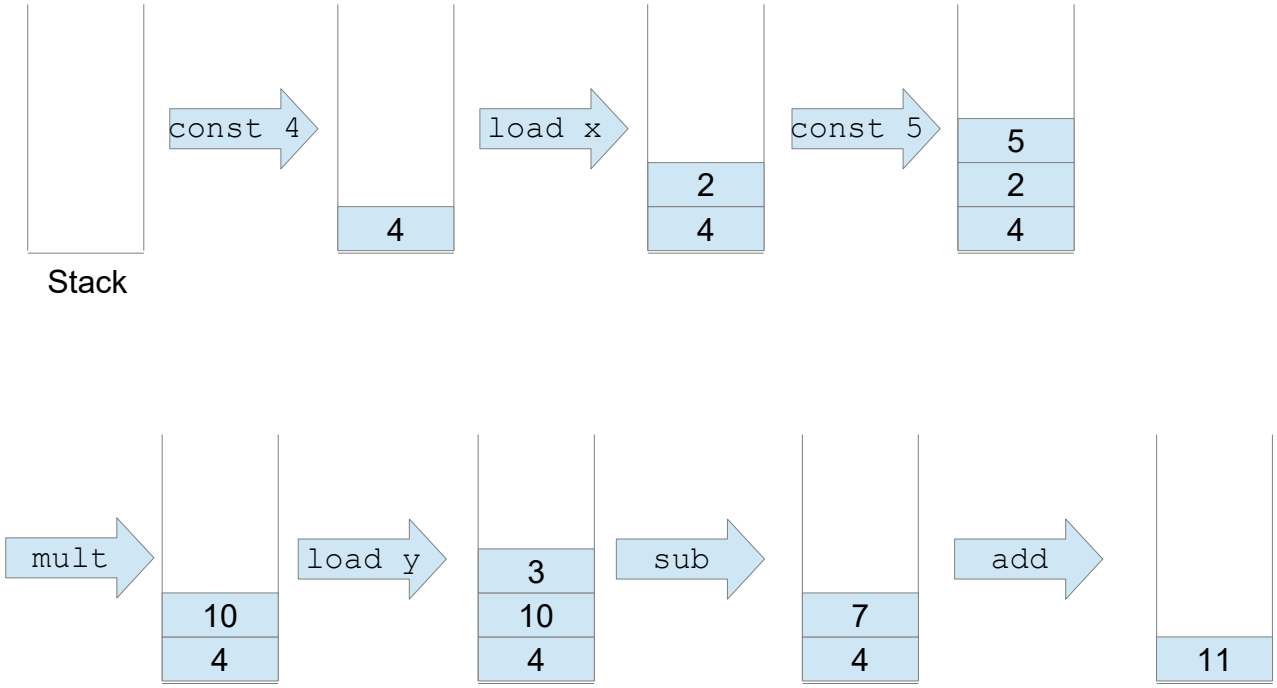
Postorder-Traversierung ergibt die Reihenfolge

```
4 x 5 * y - +.
```

Entsprechend generierter Code:

```
const 4
load x
const 5
mult
load y
sub
add
```

- **Bytecode ausführen:** Dabei wird angenommen, dass Variable `x` den Werte 2 und `y` den Wert 3 gespeichert hat.



Das Ergebnis 11 liegt am Ende oben auf dem Stack.

Aufgabe 3.63 - Bytecode-Generierung

Generieren Sie mit dem vorgestellten Verfahren Bytecode für den Ausdruck

```
(3 + x * 5)*(z - 4).
```

Fazit zu Lektion 5

Diese Fragen sollten Sie nun beantworten können

- Was sind Bäume?
- Was ist ein Spannbaum?
- Worin unterscheiden sich Bäume und Wurzelbäume?
- Was ist die Tiefe eines Knotens in einem Wurzelbaum?
- Wie ist die Höhe von Wurzelbäumen definiert?
- Wie sind Binärbäume definiert?
- Was sind vollständige Binärbäume? Welche Eigenschaften haben sie?
- Was ist Preorder-, Inorder- und Postorder-Traversierung eines Binärbaums?
- Wie kann Bytecode für eine Stack-Maschine generiert werden?