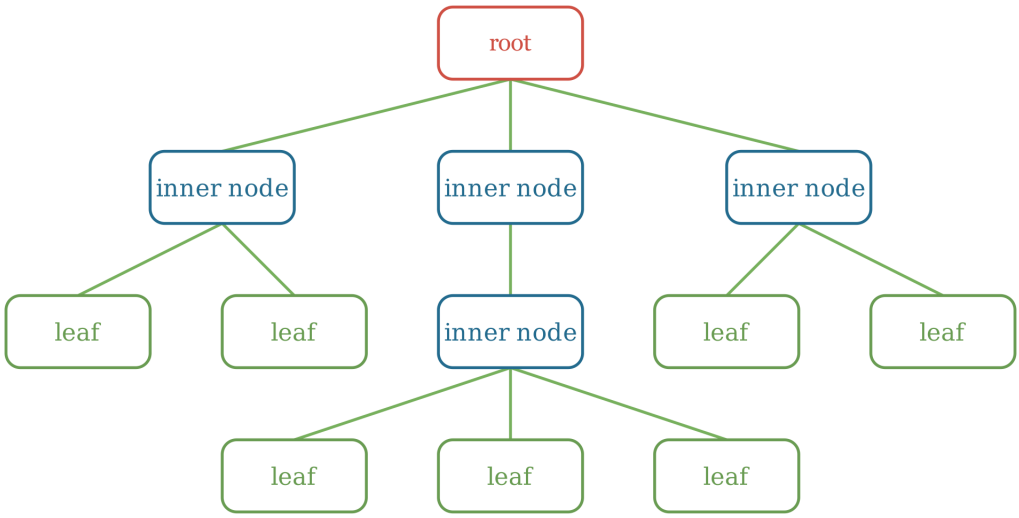
0.1 Grundbegriffe

Die Idee des Baumes in der Informatik wurde sogar schon in der 6. Jahrgangsstufe behandelt! Zur Auffrischung: Ein Baum besteht aus **Knoten** (nodes) und **Kanten** (edges). Je nachdem, wo diese Knoten liegen unterscheidet man noch zwischen der **Wurzel** (root), den **inneren Knoten** (inner node) und den **Blättern** (leaves).



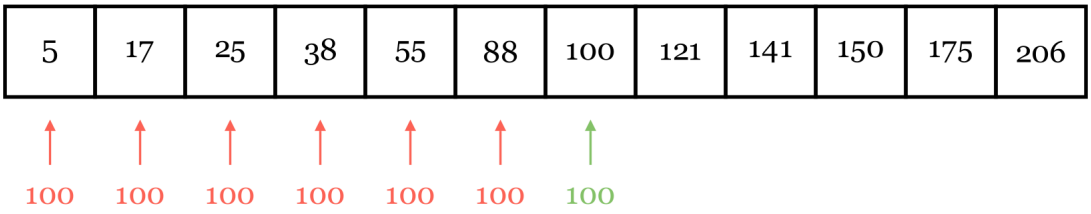
Schon die optische Betrachtung dieser Struktur vermittelt den Eindruck, das auch hier mit einer verzeigerten Struktur gearbeitet werden kann. Ähnlich wie bei der Liste gibt es eine steuerende Klasse **Baum** bzw. **Tree**, die für den Aufruf von Methoden auf der Wurzel zuständig ist.
Im Gegensatz zur Liste hat die Wurzel jetzt mehrere Nachfolger, man spricht hier auch von **Kindknoten** (**child Node**) und **Elternknoten** (**parent nodes**).
Bevor die konkrete Implementierung eine Rolle spielt, zunächst ein Beispielbereich der Informatik, in dem eine Baumstruktur sehr häufig vorkommt, die **Suche**. Dabei beschränken wir uns der Einfachheit halber zunächst auf Bäume, die **soritert** und **binär** sind, d.h. jeder Knoten kann maximal zwei Kinder haben.

0.2 Die Suche nach einer Zahl

Wir nehmen an, dass bereits eine sortierte Liste an Zahlen vorliegt, z.B.:

5	17	25	38	55	88	100	121	141	150	175	206

Die Suche nach einer gegebenen Zahl, z.B. 100 wurde bereits für die Array List und für die Linked List implementiert. In beiden Fällen besteht die grundlegende Idee darin, sich die Liste, entlangzuhangeln und an jedem Knoten (bzw. an jedem Platz im Array) zu vergleichen, ob die Zahl an dieser Stelle sitzt (für die Linked List ist dies auch in Ordnung so!).



Ist die Liste sortiert können wir zumindest noch eine Verbesserung der Performance erreichen, indem wir den Suchalgorithmus abbrechen lassen, sobald er eine Zahl erreicht, die größer als die gesuchte Zahl ist. Finden wir die Zahl, so können wir ohnehin abbrechen.

Im schlechtesten Fall (engl. Worst-Case) müssen wir aber dennoch die gesamte Liste durchlaufen und jedes einzelne Element mit unserer Eingabe vergleichen. In Bezug auf die **Laufzeit** unseres Algorithmus ist dies sicher nicht ideal, man spricht hier von einer linearen Laufzeit, oder kurz, der Algorithmus ist in $\mathcal{O}(n)$ (Details zur sogenannten $\mathcal{O}$-Notation für Interessierte im Anhang!).
Würde ein Mensch diese Suche übernehmen, so würde er sicher nicht schrittweise vergleichen, wenn bekannt ist, dass die Liste sortiert ist. Wir gehen der Einfachheit halber davon aus, dass für unseren Menschen die Liste als „Buch“ vorliegt, bei dem auf jeder Seite eine Zahl steht. Ein typisch „menschliches“ Suchmuster sähe algorithmisch etwa so aus:

```
Schlage das Buch etwa in der Mitte auf
wiederhole solange die Zahl nicht gefunden ist:
    Ist die abgebildete Zahl größer als die gesuchte Zahl:
        Suche im "linken" Bereich des Buches weiter
    sonst:
        Suche im "rechten" Bereich des Buches weiter
```

Für die Suche im „linken“ bzw. „rechten“ Bereich kann ebenso wieder die Strategie des „in der Mitte“- Aufschlagens verwendet werden. Ist man nahe an der gesuchten Zahl wird ein Mensch vermutlich auch beginnen einfach zu blättern.
In jedem Fall ist dieses Vorgehen effizienter als das bloße Durchgehen der Liste nach der Reihe, es ist eine „intelligentere“ Form des Suchens.
Wir betrachten wieder das Beispiel von oben und suchen diesmal die Zahl 150. Die Suche startet aber diesmal in der Mitte, d.h. beim 6− ten Eintrag unserer Liste , hier: 88.
(*Hinweis*: man könnte auch den Siebten Eintrag als Mitte definieren, bei geraden Anzahlen an Elementen kann auf- oder abgerundet werden, das spielt im Prinzip keine Rolle!)

5	17	25	38	55	88	100	121	141	150	175	206

Damit ist klar, dass die Zahl rechts der 88 liegen muss, die 5 Zahlen links der 88 kann ich sofort ausschließen, wir suchen uns wieder die Mitte der verbleibenden 6 Zahlen, also die 3. Zahl, in diesem Fall 141:

5	17	25	38	55	88	100	121	141	150	175	206

Da 141 immer noch kleiner als 150 ist, betrachten wir wieder den rechten Bereich und im nächsten Schritt wird mit der 175 verglichen:

5	17	25	38	55	88	100	121	141	150	175	206

Zum ersten Mal sind wir zu groß! D.h. wir müssen den linken Bereich betrachten, dieser ist jetzt aber nur noch ein Element groß! D.h. wir haben in diesem Fall die 150 gefunden:

5	17	25	38	55	88	100	121	141	150	175	206

Insgesamt hat in dieser Variante also bereits der vierte Vergleich zum Erfolg geführt, in einer linearen Implementierung wären dagegen zehn Vergleiche notwendig gewesen.
Um eine bessere Analyse des Aufwands zu erhalten, muss der durchschnittliche Suchaufwand (engl.: Average Case) betrachtet werden, um auszuschließen, dass wir in diesem Fall einfach nur „Glück“ hatten.
Um mathematisch etwas aussagen zu können, gehen wir der Einfachheit halber davon aus, dass alle Positionen

für eine Zahl im Array gleich wahrscheinlich sind- das ist im Allgemeinen keine starke Einschränkung, wenn wir nichts über die Anzahl an Zahlen und Verteilungen wissen. In unserem Beispielarray mit 12 Einträgen hat eine zu suchende Zahl also eine Wahrscheinlichkeit von jeweils $\frac{1}{12}$ an einer bestimmten Stelle zu sein.
Im besten Fall finden wir gleich beim ersten Vergleich unsere Zahl, im schlechtesten Fall (engl.: Worst Case) nach zwölf Vergleichen. Jede dieser Möglichkeiten hat aber die gleiche Wahrscheinlichkeit, deswegen können wir rechnen:

$$\frac{1}{12} \cdot 1 + \frac{1}{12} \cdot 2 + \cdots + \frac{1}{12} \cdot 11 + \frac{1}{12} \cdot 12 = \frac{1}{12} \cdot (1 + \cdots + 12) = \frac{1}{12} \cdot 78 = 6{,}5$$

Im Schnitt werden bei der **linearen Suche** also 6,5 Vergleiche gebraucht, um einen Eintrag zu finden. Zurück zu unserer **menschlichen Suche** (der Fachbegriff dafür in unserem Kontext ist eigentlich **binäre Suche** oder englisch **binary search**). Im Besten Fall finden wir auch hier nach genau einem Schritt die Lösung (wenn wir die 88 suchen würden), der schlechteste Fall (wie die 150) braucht aber nur 4 Schritte! Hätten wir beispielsweise nach **149** statt 150 gesucht, so wäre auch nach diesem vierten Vergleich die Suche beendet, da wir bereits mit der nächstkleineren Zahl, nämlich 141 verglichen hatten.
Um den durchschnittlichen Aufwand zu bestimmen kann bestimmt werden, nach wie vielen Schritten jede Zahl erreichbar ist:

5	17	25	38	55	88	100	121	141	150	175	206

Es werden:

- Eine Zahl braucht einen Vergleich
- zwei Zahlen brauchen zwei Vergleiche
- vier Zahlen brauchen drei Vergleiche
- fünf Zahlen brauchen vier Vergleiche

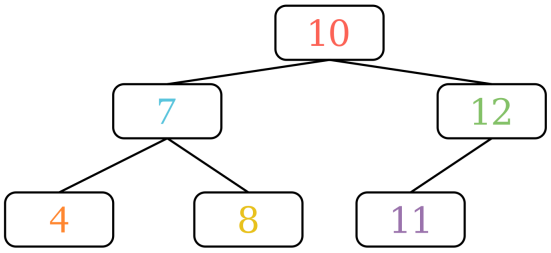
Damit ergibt sich:

$$\frac{1}{12} \cdot (1 \cdot 1 + 2 \cdot 2 + 4 \cdot 3 + 5 \cdot 4) \approx 3{,}1$$

Hier brauchen wir nur noch 3,1 Vergleiche im Schnitt. Die Verbesserung wird noch deutlicher, wenn wir große Mengen an Daten betrachten. Dazu aber später bei der Implementierung der Suchmethode mehr!

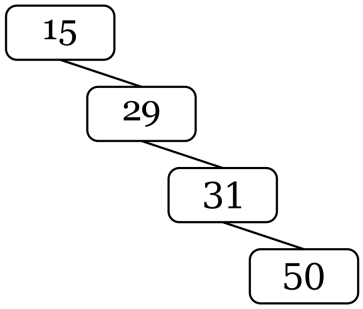
0.3 Vorüberlegung zur Implementierung

Grundsätzlich können Bäume auch in Arrays eingebettet werden, im vorherigen Kapitel haben wir z.B. die binäre Suche auch innerhalb eines Arrays ausgeführt. Dabei haben wir schon eine Baumstruktur ausgenutzt (dazu später mehr). Für die eigentliche Suche kann eine Repräsentation im Array sogar vorteilhaft sein. Eine übliche Einbettung funktioniert dabei wie folgt (i für Index, v für value):



i	0	1	2	3	4	5	6
v	10	7	12	4	8	11	

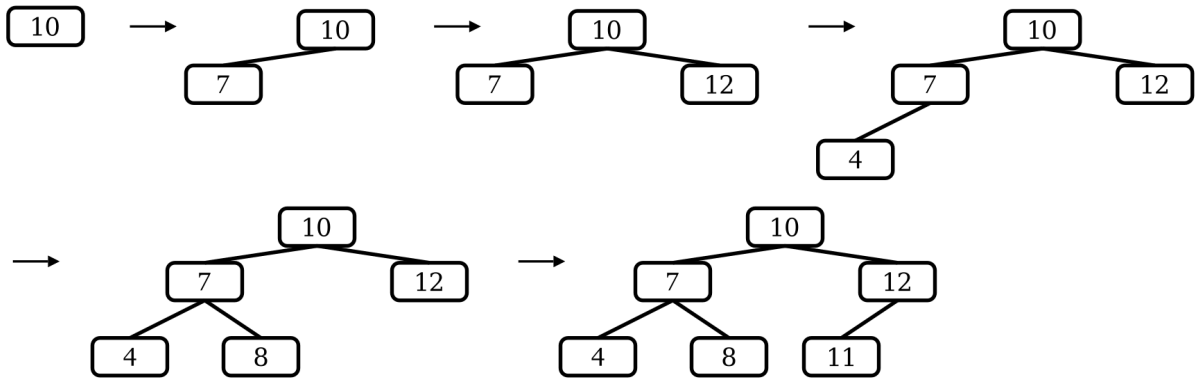
Die Wurzel liegt beim nullten Index des Arrays, in den folgenden zwei Einträgen sind die beiden Kinder platziert, in den nächsten vier Einträgen dann die vier Enkel, u.s.w..
Möchte man den Baum häufig vergrößern oder verkleinern (sprich Elemente hinzufügen oder Entfernen), so ergeben sich die selben möglichen Probleme wie bei der ArrayList! Ist der Baum darüber hinaus nicht **balanciert**, so ergeben sich viele Leerstellen.
Grob gesprochen ist ein Baum dann (aus)balanciert, wenn jede Reihe möglichst voll gefüllt ist. Der obige Baum ist beispielsweise gut balanciert. Dagegen wäre der folgende Baum sehr schlecht balanciert:



Eine andere Möglichkeit ist ein Wurzelelement - also sinnvollerweise das Element, das bei einer Suche als erstes ausgespuckt wird (in unserem Beispiel von oben also die 88) - direkt auf die weiteren Elemente verweisen, die im nächsten Suchschritt verwendet werden. Von diesem Elementen verweisen wir anschließend wieder auf die nächsten Elemente in der Suchreihenfolge, usw.
Dadurch ergibt sich automatisch die Struktur eines Baumes (wie in den Zeichnungen oben schon angedeutet!). In diesem Fall hat jeder Knoten bis zu zwei Kinder:

- der Knoten im linken Teilbaum, der verwendet wird, wenn das zu suchende Objekt „kleiner“ als das Objekt im Knoten ist.
- der Knoten im rechten Teilbaum, der verwendet wird, wenn das zu suchende Objekt „größer“ als das Objekt im Knoten ist.

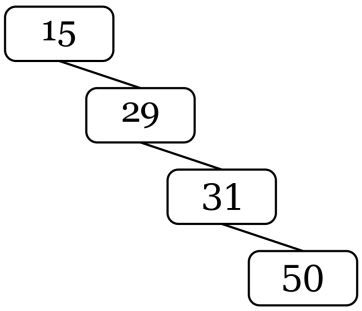
Dadurch ergibt sich auch direkt eine Möglichkeit einen binären Baum aufzubauen: bei jedem einzufügenden Element hangelt man sich am Baum entlang und wählt - abhängig vom einzufügenden Element nach obigen Kriterien - den linken oder rechten Teilbaum vom derzeitigen Knoten aus gesehen. Ist an der gewählten Seite noch kein Knoten wird dieser eingefügt.
Der Baum aus dem obigen Einbettungsbeispiel kann also beispielsweise so entstanden sein:



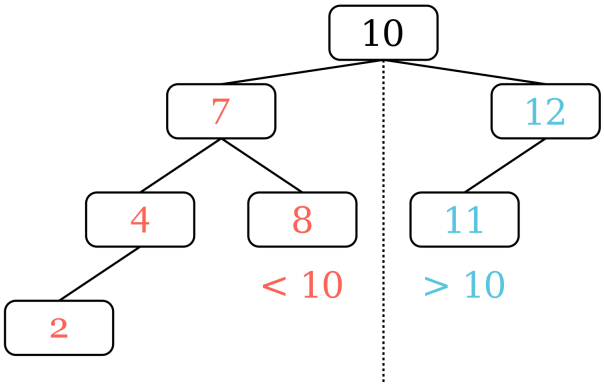
Hinweise:

- größer und kleiner bezieht sich dabei nicht notwendigerweise auf Zahlen. Wie bei unserer Liste können wir auch nach anderen Kriterien ordnen. Solange der Vergleich zweier Elemente mit dieser Methode immer ein eindeutiges Ergebnis liefert (also größer, kleiner oder gleich) können beliebige Datentypen verwendet werden.

- Die sich ergebende Struktur des Baumes hängt natürlich von der Reihenfolge des Einfügens in den Baum ab. Hat die Wurzel beispielsweise den Wert 20, wird die Zahl 22 in den rechten Teilbaum eingefügt. Hat sie dagegen den Wert 25, so wird 22 in den linken Teilbaum eingefügt.
  Dadurch kann sich auch ein „Ungleichgewicht“ ergeben. Am „schlechtesten“ ist dabei ein Einfügen in bereits sortierter Reihenfolge:



- Die Sortierung in der Baumstruktur wird noch einmal deutlich, wenn man folgendes Bild betrachtet:



  Zieht man eine Linie senkrecht unterhalb eines beliebigen Knotens, so sind alle Knoten im linken Teilbaum kleiner und alle Knoten im rechten Teilbaum größer.

Aufgabe Einfügen: Zeichnen Sie jeweils schrittweise den binären Suchbaum, der sich ergibt, wenn die folgenden Zahlen in der gegebenen Folge eingefügt werden.
a) $5 \to 7 \to 3 \to 2 \to 4 \to 6 \to 10 \to 11$
b) $10 \to 11 \to 5 \to 7 \to 6 \to 4 \to 3 \to 2$

Und jetzt auf zur Implementierung!