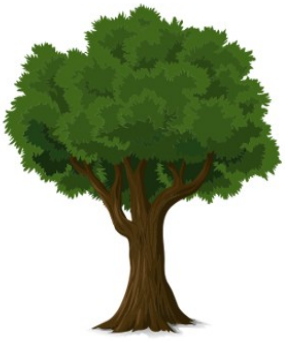
Datenstruktur B A U M



Die Datenstruktur Liste ist dynamisch und sehr flexibel. Unpraktisch wird sie allerdings, wenn ein Element innerhalb der Liste gesucht wird. Werden mehrere Millionen Datensätze in einer sortierten Liste verwaltet, liegt der „Worst case“ vor, wenn gerade das letzte Element in der Liste vom Benutzer gesucht wird. Dann muss die ganze Liste durchlaufen werden, d. h. jedes Element ausgelesen werden, um das gesuchte Element zu finden. Bei einem Baum darf jeder Knoten beliebig viele Nachfolger besitzen. Beispiel: Dateisystem

Die Datenstruktur ***B a u m*** bietet eine schnellere Lösung!

Einsatzbereich eines Binärbaums

Betrachten wir zunächst noch einmal kurz die Datenstruktur LISTE und darin unsere bisher implementierte Methoden einfügenSortiert*(...)*. Beide haben bisher gute Dienste geleistet. Wie verhalten sich diese Methoden aber, wenn wir sehr viele Datensätze verwalten. Nehmen wir z. B. die Datenverwaltung von *Instagram* oder *WhatsApp*. Hier werden mehr als 1 Milliarde (1.000.000.000) Datensätze verwaltet.

Aufgabe 2:

Stelle dir die beiden Extremfälle einer Suche nach einem Nutzer mit der Kennung **a** im Vergleich zur Suche nach dem Nutzer mit der Kennung **zzzzzzzzzz** bei 1 Milliarde Datensätze vor.

a) Was geschieht jeweils innerhalb der Liste, bis der Nutzer gefunden wird.

b) Nehmen wir an, ein Vergleich auf dem Server dauert 5µs und ein Referenz-Wechsel auf den Nachfolger 1µs.
Wie lange dauert dann vermutlich das Finden von aaaaaaaaa?
Wie lange dauert dann vermutlich das Finden von zzzzzzzzzz?
Wie lange dauert dann im Mittel eine Suche?

c) Bearbeite Kapitel 1: Eine Liste umstrukturieren von JustTrees

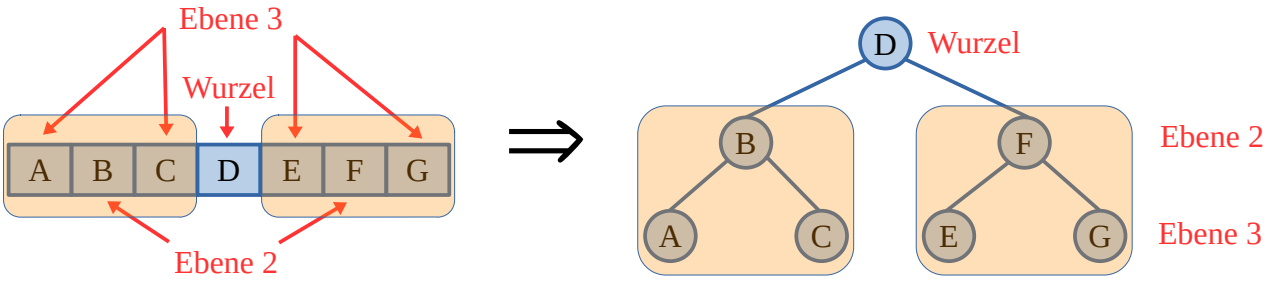
d) Zusammenfassung: Wie könnte man vorgehen, sodass die Suche in der Liste im Mittel erheblich schneller stattfinden kann? Beschreibe kurz deinen Algorithmus.

Ein mögliches Verfahren zur schnelleren Suche in einer Liste könnte folgendermaßen vorgehen:

- Nimm das mittlere Glied der sortierten Liste. Bei geradzahliger Länge der Liste nimm z.B. den linken der beiden mittleren Knoten.

- Vergleiche den Such-String mit der Info dieses Knotens. War die Suche erfolgreich, so gib *true* zurück. Ist der Such-String jedoch „größer als“ die Info, so verfahre auf dieselbe Art mit der rechten Teilliste. War der Such-String „kleiner als“ die Info, so mache mit der linken Teilliste entsprechend weiter.
- Besteht eine Teilliste nur noch aus einem Element und entspricht dessen Info nicht dem Such-String oder hat eine Teil-Liste die Länge 0, so gib *false* zurück.

In einem (balancierten) Binär-Baum werden die Knoten nun genau in der Reihenfolge erzeugt, wie man bei der Suche in einer sortierten Liste vorgehen würde. Das folgende Beispiel soll das veranschaulichen:



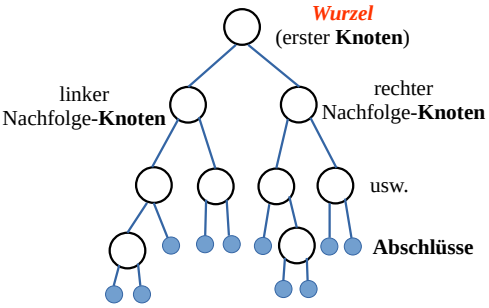
Eine ***Ordnung in einem Binärbaum*** wird also dadurch realisiert, dass ausgehend von einem beliebigen Knoten

- alle Elemente im ***linken Teilbaum „kleiner als“*** das Element im darüber liegenden Knoten sind.
- alle Elemente im ***rechten Teilbaum „größer als“*** das Element im darüber liegenden Knoten sind.

Hierbei wird vereinfachend davon ausgegangen, dass wir nach einem eindeutigen Schlüssel (ähnlich dem Primärschlüssel bei einer Datenbanktabelle) sortieren, dass es also keine doppelten Elemente gibt. Das Einfügen eines „doppelten Datensatzes“ sollte also zu einer Fehlermeldung führen!

Datenstruktur B I N Ä R B A U M

Ein ***Binärbaum*** ist grundlegend genauso aufgebaut wie eine Liste, nur dass jeder Knoten bis zu zwei Nachfolger haben kann. Deshalb spricht man auch von einem linken und einem rechten Nachfolger. Ein Knoten kann also effektiv keine, nur einen linken, nur einen rechten oder zwei Nachfolger haben.



Man kann einen beliebigen Knoten und dessen Nachfolger (und deren Nachfolger …) aus einem Binärbaum herausnehmen und hat wieder einen Binärbaum vor sich. → ein beliebiger Teil des Ganzen wieder exakt dieselbe Struktur wie das Ganze selbst → Leitidee des Entwurfsmusters Kompositum

- Den ersten Knoten – den, der keine Vorgänger hat – nennt man ***wurzel*** *(engl. root)*.
- Knoten, die wenigstens einen Nachfolger haben, nennt man ***innere Knoten***.

- Knoten, die keine Nachfolger mehr haben, nennt man ***Blatt***.
- Die Verbindungslinien zu den Nachfolgeknoten nennt man ***Kante***.
- Sucht man sich einen Weg von der Wurzel zu einem bestimmten Knoten, so spricht man von einem ***Pfad***.
  Ein Pfad beginnt immer bei der Wurzel (w) und geht dann mit links (l) oder rechts (r) weiter bis zu einem bestimmten Knoten.
  Beispiel: *Der Pfad zum Knoten ganz links unten in obigem Bild lautet: w l l l .*
- Die Anzahl der Knoten im längsten möglichen Pfad nennt man ***Höhe des Baums***.
  *(**Achtung:** Pfade gehen maximal bis zu einem Blatt, nie bis zu einem Abschluss!)*
- In der Zeichnung nebeneinander liegende Knoten nennt man eine ***Ebene***. Die Ebene der Wurzel nennt man **Ebene 1**, die Ebene der Nachfolger der Wurzel ist **Ebene 2**, usw.

Aufgabe 1:

a) Welche Höhe hat der Baum in obigem Bild?

b) Wie viele Knoten befinden sich bei obigem Baum in Ebene 4?

c) Wie viele Knoten passen maximal in die Ebene 1, 2, 3, 4, … ? Stelle eine allgemeine Formel in Abhängigkeit der Höhe auf.

d) Wie viele Knoten passen insgesamt maximal in einen Binärbaum der Tiefe 1, 2, 3, 4, … ? Stelle eine allgemeine Formel in Abhängigkeit der Höhe auf.

e) Gib den Pfad zum rechten Knoten der untersten Ebene in obigem Bild an.

f) **Zeichne ein Klassendiagramm eines Binärbaums mit Kompositum.**

g) - Implementiere dieses Klassendiagramm in JAVA.
- Erstelle hierzu alle nötigen Klassen und Referenzattribute.
- Die Konstruktoren sollen die Referenzattribute bereits initialisieren.
- Darüber hinaus kannst du bereits alle Setter- und Getter-Methoden implementieren.
- Das Interface soll wie bei der Liste eine Methode `info() : String` sowie Vergleichsmethoden vorschreiben.

Anzahl an Vergleichen bei Suchoperationen (Binärbaum vs. Liste)

Beispiel:

In einem Binär-Baum werden 10 000 Elemente gespeichert. Welche Ober- bzw. Untergrenze für die Anzahl der maximal nötigen Vergleiche, die zum Suchen eines Elements nötig sind, ergeben sich?

a) „Worst-Case“: **der Baum ist zu einer Liste *entartet***.

→ Es benötigt im Maximalfall 10 000 Vergleiche das Element zu Suchen.

b) „Best-Case“: **der Baum ist *balanciert* und *vollständig* befüllt**, d.h. alle Ebenen sind voll ausgefüllt.

→ Im schlimmsten Fall muss ein Pfad von der Wurzel bis zum Blatt durchlaufen werden. D.h. die Tiefe entspricht der Anzahl an nötigen Vergleichen.

Wir wissen:
$2^h - 1 = n$, wobei **h** = Höhe (Tiefe) und **n** = Anzahl an enthaltenen Knoten.

Daraus ergibt sich:

$2^h - 1 = 10\,000$ | + 1

$2^h = 10\,001$ | $\log_2(...)$

$h = \log_2(10\,001) \approx 13{,}29$

→ es sind im Maximalfall bei einem optimalen Baum 14 Vergleiche nötig.

Aufgabe 3:

a) Zeichne einen Binärbaum, in den die Buchstaben M, D, X in genau dieser Reihenfolge eingefügt wurden.

b) Wo käme beim Baum aus Aufgabe a) der Buchstabe P hin, wenn er nachträglich eingefügt werden soll?
Zeichne ein Sequenzdiagramm, welches alle nötigen Schritte veranschaulicht, die nötig sind, um P einzufügen.

c) **Implementiere die Methode** einfuegen(T t) : void**.**

Ein Knoten prüft erst, ob er das Problem an den linken oder rechten Nachfolger delegieren muss und setzt seinen entsprechenden Nachfolger dann neu auf die Antwort dieses Nachfolgers.

Aufgabe 4:

a) Formuliere eine Strategie, wie man in einem Binärbaum suchen kann, ob sich schon ein Element mit einem bestimmten Info-String darin befindet.
Betrachte den Binärbaum aus Aufgabe 3a) und zeichne ein Sequenzdiagramm, welches die Suche nach „F“ beschreibt.

b) **Implementiere die Methode** istEnthalten(String s): boolean**.**

Das Vorgehen ähnelt dem des Einfügens:

Der Binärbaum delegiert das Problem an seine Wurzel weiter.

Ein Knoten gibt **`true`** zurück, wenn er ein Element mit dem gesuchten Info-String enthält. Ansonsten delegiert er die Suche an seinen linken bzw. rechten Nachfolger, je nachdem ob der Such-String kleiner oder größer als der eigene Info-String ist. Da String kein primitiver Datentyp ist, funktionieren Vergleiche auf Strings mit der Methode String.compareTo(String s). Siehe Java-API-Doc.

Ein Abschluss gibt immer **`false`** zurück.

c) **Ändere nur die Methodensignatur im Binärbaum zu** einfuegen(T t): boolean **um, sodass zuerst geprüft wird, ob sich das einzufügende Element nicht schon im Binärbaum enthalten ist. Ist das der Fall, so gib** `false` **zurück, andernfalls füge das Element (wie vorher) ein und gib** `true` **zurück.**

d) Optional: Implementiere die Methode **`suchenUndZurückgeben(String s): T`**

Aufgabe 5:

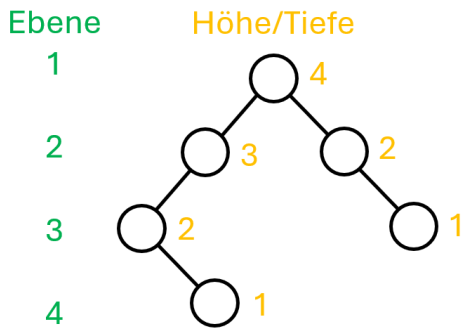
a) Formuliere eine Strategie, wie man bei einem Binärbaum die (maximale) Höhe herausfinden kann.
Zeichne einen Binärbaum, in den der Reihe nach „M“, „F“, „R“ und „V“ eingefügt werden und teste händisch deinen Algorithmus, ob er das Gewünschte leistet.

b) **Implementiere nun die Methode** höhe(): int**.**

Der Binärbaum delegiert die Methode an seine Wurzel weiter.

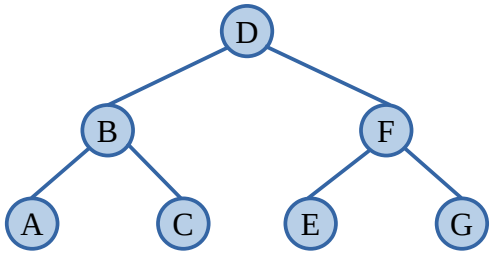
Ein Knoten fragt erst seinen linken und rechten Nachfolgebaum nach dessen Tiefe. Zum größeren der beiden Werte (**Math.max(zahl1, zahl2, ...)**) addiert er Eins (der Knoten selbst steht ja noch über den beiden Nachfolgebäumen) und gibt diesen Wert zurück.

Ein Abschluss gibt generell 0 zurück.



c) Optional: Implementiere die Methode **`sucheUndGibEbene(String s): int`**

Traversierung eines Binärbaums



Um z.B. alle Elemente eines Binärbaums zu zählen braucht man einen Algorithmus, der sicher über alle Knoten läuft und dabei keinen auslässt oder mehrfach zählt. Sprich: Unter dem Begriff ***Traversierung*** versteht man, dass ein Baum derartig durchlaufen wird, so dass **jeder Knoten** dabei **genau einmal** erreicht wird. Es gibt verschiedene Arten, dies zu tun:

1. ***in-order-Durchlauf***
   durchlaufe den linken Teilbaum, dann den Knoten selbst und anschließend den rechten Teil-Baum => *(**in**-order: **in der Reihenfolge** von links nach rechts)*

   Vorteil des Verfahrens: Der Binärbaum wird sortiert ausgegeben.
   Allerdings ist dieses Verfahren nicht zum Abspeichern geeignet, da beim Einlesen in dieser Form ein entarteter Baum in Form einer Liste entstehen würde.
   Am Beispielbaum angewendet ergibt dies folgende Ausgabe: A, B, C, D, E, F, G

2. ***prä-order-durchlauf***
   durchlaufe zuerst den Knoten selbst, dann den linken Teilbaum und anschließend den rechten Teilbaum
   *(prä-order: der Knoten selbst kommt **vor** dem linken und rechten Teilbaum)*

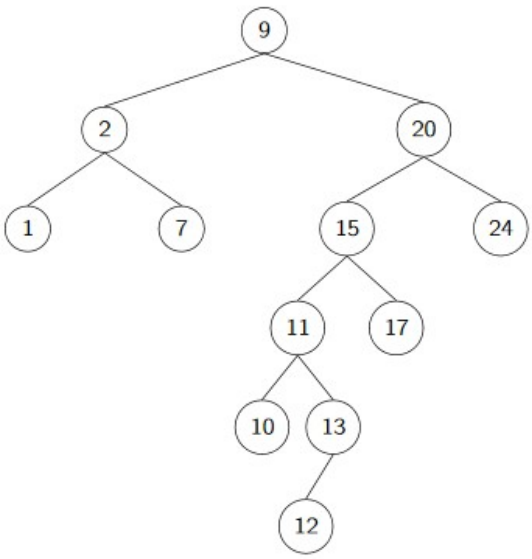
   Dieses Verfahren eignet sich besonders zum Abspeichern von Binärbäumen, denn wenn man die Elemente wieder einliest, erhält man den gleichen Baum.
   Am Beispielbaum angewendet ergibt dies folgende Ausgabe: D, B, A, C, F, E, G

3. ***post-order-Durchlauf***
   durchlaufe den linken Teilbaum, dann den rechten Teil-Baum und anschließend den Knoten selbst
   *(**post**-order: der Knoten selbst kommt erst **nach** dem linken und rechten Teilbaum)*

   Dieses Verfahren wird in der Praxis eher weniger gebraucht, wird aber zur Vollständigkeit mit aufgeführt.
   Am Beispielbaum angewendet ergibt dies folgende Ausgabe: A, C, B, E, G, F, D

Dabei wird der **Algorithmus** jeweils **rekursiv** auf den linken bzw. rechten Teilbaum angewendet.

Beim Kompositum wird die **Abbruchbedingung** wieder in der Klasse *Abschluss* realisiert.

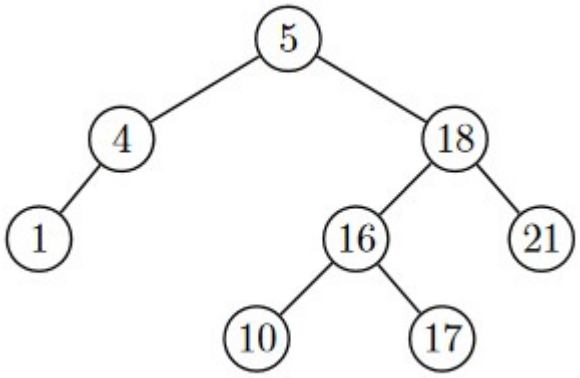


Aufgabe 6:

a) Betrachte die beiden untenstehenden Bäume.

Gib seine Elemente händisch einmal in-order, einmal prä-order und einmal post-

order aus.



b) **Implementiere nun die Methoden** inOrder(): void **,** praeOrder(): void **,** postOrder(): void **indem du den Info-String jedes Elements auf der Konsole mit `System.out.println(...)` ausgeben lässt.**

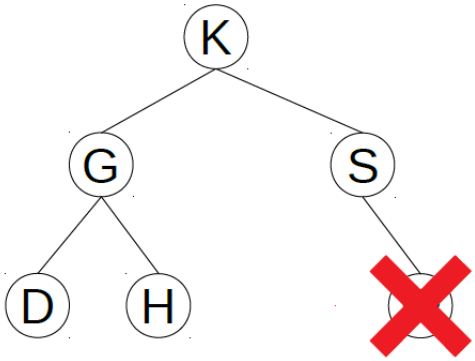
c) **Implementiere zusätzlich eine Methode** anzahlElemente(): int**, um die Anzahl an Elementen des Baums zu bestimmen.**

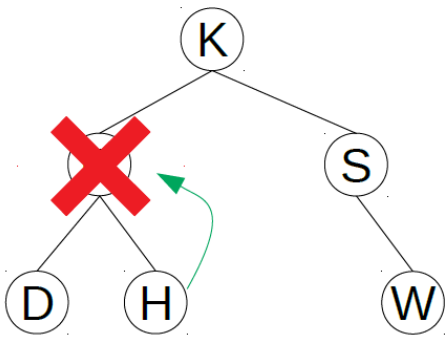
Löschen aus einem Binärbaum

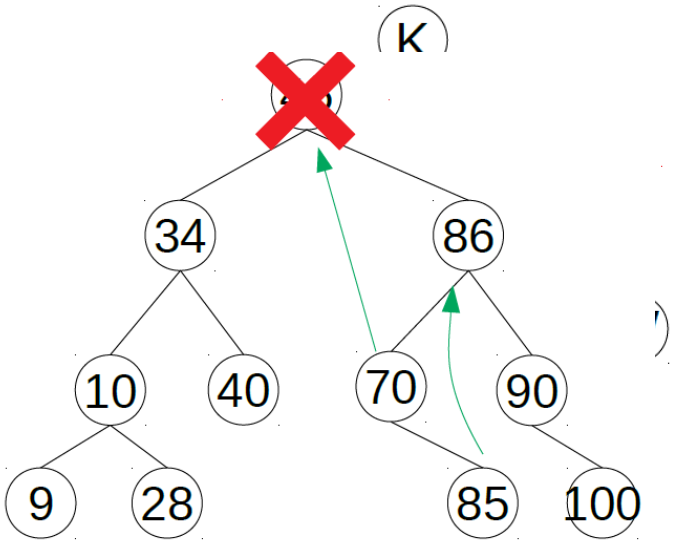
Sollen Elemente wieder aus einem Binärbaum gelöscht werden, so unterscheidet man drei Fälle:

1. Fall: Blattknoten

Der zu löschende Knoten ist ein Blatt und besitzt ***keinen Nachfolger***. Somit kann der Knoten ohne Weiteres aus dem Baum gelöscht werden. Die Referenz wir auf einen Abschluss gesetzt.







2. Fall: Innerer Knoten mit nur einem Nachfolger

In diesem Fall wird die Referenz auf den zu löschenden Knoten einfach durch dessen Nachfolger ersetzt.

3. Fall: Innerer Knoten mit zwei Nachfolgern

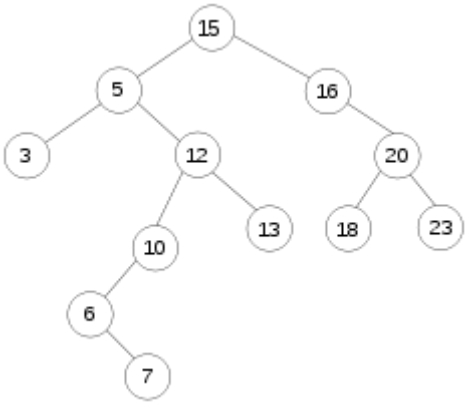
Nun kann der Knoten nicht einfach gelöscht werden! Es gibt nur eine Referenz auf den zu löschenden Knoten, dieser hinterlässt nach dem Löschen aber zwei Referenzen auf seine Nachfolger. Was nun?

Kurzfassung: In diesem Fall wird der Knoten, der in der Inorder-Traversierung unmittelbar auf den zu löschenden Knoten folgt (sprich der nächstgrößere Knoten), an dessen Stelle gesetzt. Hierbei ergibt sich wieder einer der Fälle 1 – 2.

Versuche zunächst, die Richtigkeit dieses Algorithmus (ohne zu programmieren) an einigen Beispielen zu überprüfen:

Aufgabe 7:

Betrachte den folgenden Binärbaum:



a) Teste den **Fall 1** an den Knoten mit dem Inhalt 3 und 7.

b) Teste den **Fall 2** an den Knoten mit dem Inhalt 6, 10 und 16.

c) Teste den **Fall 3** am Knoten mit dem Inhalt 5. Wähle das Nächstkleinere.

d) Teste den **Fall 3** an den Knoten mit dem Inhalt 5. Wähle das Nächstgrößere.

e) Teste den **Fall 3** am Knoten mit dem Inhalt 15.

Wähle einmal das Nächstkleinere und einmal das Nächstgrößere.

Programmiertechnische Umsetzung von Fall 3:

Die Lösung besteht darin, den Knoten mit dem nächstgrößeren (kleineren) Inhalt zu suchen und diesen Inhalt zurückzugeben. Man findet diesen Inhalt nach folgendem Algorithmus:

- **nächstgrößeres Element**
  Gehe zum rechten Nachfolger und von dort aus so lange rekursiv nach links, wie es einen linken Nachfolger gibt. Übergebe jeweils die Daten (die Auflistbarreferenz) des aktuellen Knoten und der der Abschluss gibt den letzten Inhalt zurück.
- **nächstkleineres Element**
  Gehe zum linken Nachfolger und von dort aus so lange nach rechts, wie es einen rechten Nachfolger gibt. Übergebe jeweils die Daten des aktuellen Knoten und der der Abschluss gibt den letzten Inhalt zurück.

Anschließend kopiert man den Inhalt dieses Knotens in den eigentlich zu löschenden Knoten und löscht stattdessen den Knoten mit dem nächstgrößeren (kleineren) Inhalt. Dieser ist löschbar, da für diesen Knoten mit dem nächstgrößeren (kleineren) Inhalt der Fall 1 oder 2 gilt.

Aufgabe 8:

Implementiere die Methode entferne(String s) : boolean.

Tipps:

1) Es ist sinnvoll, in den Klassen Element/Knoten/Abschluss zwei Methoden **naechstGroesseres(T t): T** und **naechstKleineres(T t): T** einzuführen, mit denen man eine Referenz auf den Inhalt des Knotens mit dem nächstgrößeren (kleineren) Element erhält.
Benutzung: Rufe wie in der obigen Anleitung beschrieben auf seinem ***rechten Nachbarn*** die Methode `naechstGroesseres(null)` auf bzw. auf seinem ***linken Nachbarn*** die Methode `naechstKleineres(null)` und es sollte das jeweilig passende größere/kleinere Element im Baum ausgegeben werden.

2) Die Methode **entferne(String s)** in der **Klasse BINBAUM** gibt einen **boolean** zurück:
true, wenn das Element im Baum gefunden und gelöscht wurde;
false, wenn es im Baum kein solches Element gibt;
Der Baum setzt seine Wurzel neu auf die Antwort der Wurzel, wenn er sie mit der Löschung beauftragt.

3) In der **Klasse KNOTEN** gibt die Methode **entferne(String s)** das **Element** zurück, welches das aufrufende Objekt als seinen linken bzw. rechten Nachfolger (oder der Baum als Wurzel) setzen soll (Kompositum).
Es wird zuerst überprüft, ob der Knoten selbst gelöscht werden soll.

Wenn JA, dann arbeitet man die Fälle „kein Nf“, „nur liNf“, „nur reNf“, „2 Nf“ ab. Bei zwei Nachfolgern gibt man das Problem an einen Nachfolgebaum weiter, nachdem der neue Datensatz in diesen Knoten kopiert wurde.
Wenn NEIN, dann setzt man abhängig vom Entferne-String den linken oder rechten Nf auf das, was dieser antwortet, wenn er den Löschauftrag bekommt (Kompositum) und gibt sich selbst zurück.

4) Optional: Wann nimmt man das nächstgrößte oder -kleinste Element?
Man könnte sich stets so entscheiden, dass der Löschauftrag an den tieferen der beiden Nachfolgebäume weiter gereicht wird.
(Somit gleicht der Baum unterschiedliche Tiefen der Nachfolgebäume beim Löschen zumindest ein wenig aus.)

Nicht mehr Teil des Schulstoffs: Balancierte Bäume (AVL-Bäume)

Aufgabe 9:

Implementiere zum Thema BINÄRBAUM Abituraufgaben!

Mebis-Link zum Prüfungsarchiv aller Informatikabituraufgaben

Schülerlösungsvorschläge zu den Abituraufgaben vom Rupprecht-Gymnasium