Algorithmen & Datenstrukturen: Binärer Baum

Klaus Kusche

Gesucht ist ein C++-Programm, das einen ***binären Suchbaum*** implementiert.

Unser Baum soll *zählen*, *welche Worte wie oft* in der Eingabe vorkommen.
Unsere Baumknoten haben daher einen **`string`** (das jeweilige Wort) als *Sortierschlüssel*
und einen **`int`** (die Anzahl seiner Vorkommen) als *zweites Nutzdatum*.

Zuerst brauchen wir eine *Klasse* **Node** für die *einzelnen Knoten* unseres Baumes.
Sie soll *ausschließlich für unsere Baum-Klasse verwendbar* sein und enthält:

- *Vier Member*: Einen **`string`** für das *Wort*, einen **`int`** für seine *Anzahl*,
  und die *beiden Pointer auf die Söhne*.
- Einen *Konstruktor* mit einem **`string`**-*Parameter*.
  Er speichert den String im neuen Knoten und setzt seine *Anzahl* auf **`1`**.
  Ein neuer Knoten wird *immer als Blatt* an den Baum gehängt.
  Wie initialisierst du daher die *Sohn-Pointer*?

Set- oder Get-Methoden brauchen wir erst in den Zusatzaufgaben,
die Baum-Klasse darf ja ohnehin direkt auf die Knoten-Member zugreifen.
Baumknoten sollen *nie kopiert und nie per Kopie zugewiesen* werden:
Wir kopieren oder speichern immer nur die Pointer darauf.
Daher *verbieten* wir den *Copy-Konstruktor* und den *Zuweisungs-Operator*.

Weiters schreiben wir für unsere Knoten-Klasse gleich *zwei Ausgabe-Operatoren*.
Beide sollen *direkt auf die Member* der Baum-Knoten zugreifen können.

- Der erste Ausgabe-Operator wird mit einem *Knoten-Objekt* (*"by Reference"*
  übergeben) aufgerufen. Er gibt *"wort: anzahl"* aus (ohne Zeilenvorschub).
- Der zweite Ausgabe-Operator wird mit einem *Pointer auf einen Baumknoten*
  aufgerufen. Bei einem *Nullpointer* gibt er *gar nichts* aus.
  Sonst gibt er *rekursiv den gesamten Teilbaum* ab diesem Knoten aus,
  und zwar *zeilenweise in sortierter Reihenfolge*.

Dann kommt die *Klasse* **`Tree`** für *ganze Bäume*.

Wir werden zur Implementierung der öffentlichen Funktionen dieser Klasse
(auch für die Zusatzaufgaben) mehrere *interne, rekursive Hilfsfunktionen* brauchen.

- Alle diese rekursiven Hilfsfunktionen haben einen *Pointer*
  auf den obersten Knoten des zu bearbeitenden *Teilbaumes* als *Parameter*.

  Achtung: Alle diese rekursiven Hilfsfunktionen müssen auch richtig reagieren,
  wenn sie für einen *leeren Teilbaum (Rekursionsende!)* aufgerufen werden!
- Wir schreiben diese Hilfsfunktionen alle *innerhalb* unserer Klasse Tree,
  und zwar so, dass sie *von außen nicht aufrufbar* sind
  (unser Klassen-interner Hilfscode geht ja Außenstehende nichts an).

  Allerdings sind standardmäßig alle Funktionen innerhalb einer Klasse Methoden,
  d.h. sie werden *für ein Objekt* aufgerufen und haben ein **`this`**
  und automatisch *Zugriff auf die Member* dieses Objektes.

Unsere rekursiven Hilfsfunktionen sind aber *normale Funktionen, keine Methoden*:
Sie brauchen *kein* **this** und auch *keinen Zugriff* auf das Wurzelpointer-Member des aufrufenden Baum-Objektes.
Wie deklarierst du diese Funktionen, damit dieser **this**-Overhead wegfällt?

Die Klasse **Tree** enthält:

- Ein rein internes *Member*: Den *Pointer auf den Wurzel-Knoten*.
- Einen *Konstruktor ohne Parameter*, der einen *leeren* Baum anlegt.
- Vorläufig *keinen Copy-Konstruktor* und *keinen Zuweisungs-Operator* (*verbieten*!).
- Einen *Destruktor*, der beim Löschen eines Baum-Objektes auch *alle darin enthaltetet Knoten-Objekte* freigibt.
  Schreib dafür eine *rekursive Hilfsfunktion*.
- Eine *Methode* **count** *ohne Returnwert* und mit einem **string** als *Parameter*.
  Sie soll *ein Vorkommen* des als Parameter übergebenen Wortes *zählen*.

  Dazu muss sie das Wort zuerst einmal *im Baum suchen* (mit einer Schleife, *nicht rekursiv*, so wie wir das gelernt haben von der Wurzel abwärts mit Links-oder-Rechts-Entscheidungen).

  Findet sie das Wort im Baum, so *erhöht* sie dessen Zählerstand um 1.
  Findet sie es *nicht*, hängt sie einen *neuen Knoten* mit diesem Wort an den Baum an (neue Knoten bekommen vom **Node**-Konstruktor automatisch Zählerstand 1), und zwar *an der Stelle* des Baumes, wo sie *weitergesucht* hätte (bzw. als neue Wurzel, wenn der Baum noch komplett leer war).

Weiters schreiben wir einen *Ausgabe-Operator* für ein **Tree**-Objekt ("by Reference"!).
Er soll direkt auf das Wurzelpointer-Member zugreifen können
und ruft einfach den **Node**-Pointer-Ausgabe-Operator für die Baumwurzel auf.

Diesmal wollen wir "echte" Worte zählen, also *Zeichenfolgen nur aus Buchstaben*:
Ziffern, Sonderzeichen usw. trennen Worte und sollen *ignoriert* (überlesen) werden.
Wir schreiben für unser Hauptprogramm eine **Hilfsfunktion** (außerhalb aller Klassen), die ein solches Wort von **cin** einliest. Diese Hilfsfunktion soll das gelesene Wort in einem **string**-*Parameter* *speichern* (wie musst du den Parameter daher übergeben?) und *Erfolg oder Misserfolg* über den *Returnwert* anzeigen (**true** / **false**).

Die Funktion arbeitet wie folgt:

- Zuerst wird der Ergebnis-String auf *leer* gesetzt
  (Das kannst du entweder mit einer Zuweisung oder der Methode **clear** machen).
- Dann wird in einer *Schleife* immer wieder *ein einzelnes Zeichen* von **cin** in eine **char**-Hilfsvariable *gelesen*. Nimm dafür *nicht* **>>**
  (denn **>>** *ignoriert* Zwischenräume, Zeilenvorschübe usw.),
  sondern die Methode **get** mit der Hilfsvariable als Parameter
  (die nichts überliest, sondern wirklich alle Zeichen aus der Eingabe liefert).
- *Prüfe* in der Schleife mit **isalpha** (**isalpha** ist eine Funktion aus **ctype.h** bzw. **cctype**), ob das gelesene Zeichen *ein Buchstabe* ist.
  - Wenn *ja*, wird es an den Ergebnis-String *angehängt* (mit **+=**).
  - Wenn *nein*, prüfe, ob der Ergebnis-String *noch leer* ist

(entweder mit einem Vergleich oder mit der Methode **`empty`**).

Ist er *nicht leer*, so ist das aktuelle Zeichen der erste nicht-Buchstabe *nach* dem soeben gelesenen Wort, und die Funktion *kehrt sofort mit Erfolg zurück*.

Ist der String noch *leer*, *ignorieren* wir das soeben gelesene Zeichen und *lesen weiter* (in der Hoffnung, dass irgendwann einmal ein Buchstabe kommt, d.h. ein Wort beginnt).

- *Scheitert das Einlesen* eines Zeichens von **`cin`** (z.B. wegen Dateiende des Inputs), *endet* die Schleife (du kannst das **`get`** direkt als *Schleifenbedingung* verwenden).

  In diesem Fall müssen wir *nach der Schleife* prüfen, ob der Ergebnis-String *leer* ist. Ist er noch *leer*, endet die Funktion mit *Misserfolg*, enthält er *Text*, liefert sie *Erfolg*.

Lass dich nicht verwirren, dass **`isalpha`** *Umlaute nicht als Buchstaben* betrachtet. Beispielsweise wird daher das Wort *"zerfällt"* als zwei Worte *"zerf"* und *"llt"* gelesen, aber das soll uns nicht stören.

Das **Hauptprogramm**

- legt ein **`Tree`**-*Objekt* an (*als lokale Variable*, nicht dynamisch),
- *liest* in einer *Schleife* mittels unserer *Hilfsfunktion* immer wieder ein Wort ein (die Schleife soll laufen, bis unsere Hilfsfunktion Misserfolg meldet) und *zählt* es in unserem Baum durch Aufruf der Methode **`count`**,
- und gibt den Baum *nach der Schleife* mittels *Ausgabe-Operator* aus.

Zusatzaufgabe 1: Grafische Baum-Ausgabe

Wir erweitern unsere Klasse um eine *Methode* **`printTree`** (*ohne* Parameter und *ohne* Returnwert), die einen *Baum in grafischer Darstellung ausgibt*.
Am einfachsten geht das *liegend mit der Wurzel ganz links*,
d.h. mit *einem Wort pro Zeile* wie in unserer bisherigen Ausgabe,
aber mit *Einrückung je nach Tiefe* im Baum und mit *Verbindungs-Strichen*.

Dazu brauchen wir eine *rekursive Hilfsfunktion*.
Diese hat *keinen Returnwert* und *drei Parameter*:
- Einen *Pointer* auf den obersten Knoten des auszugebenden *Teilbaumes*.
- Einen **`bool`**, der anzeigt, ob dieser Teilbaum *oberhalb* (**`false`**) oder *unterhalb* (**`true`**) *seines Vaters* ausgegeben wird.
- Einen **`string`** mit der *Einrückung*, die vor jeder Zeile des Teilbaumes auszugeben ist.

  - Ist der Teilbaum *leer*, kehrt die Funktion sofort zurück.
  - Ist die *Einrückung* schon *länger als 70 Zeichen*, wird statt dem Teilbaum nur eine Zeile mit der Einrückung und "**`+--...`**" ausgegeben.
  - Sonst wird zuerst einmal der *linke Unterbaum* rekursiv ausgegeben. Für seine *Einrückung* wird die eigene Einrückung plus "**`|`** " verwendet, wenn wir unterhalb unseres Vaters liegen, und sonst Einrückung plus " ".
  - Dann wird der *aktuelle Knoten* ausgegeben, und zwar mit "**`---`**" davor, wenn die Einrückung leer ist (denn dann ist er die Wurzel des gesamten Baumes), und sonst mit Einrückung plus "**`+--`**" davor.
  - Zuletzt wird der *rechte Unterbaum* rekursiv ausgegeben.

Seine *Einrückung* ist Einrückung plus " ", wenn wir unterhalb unseres Vaters liegen, oder wenn die eigene Einrückung leer ist, und Einrückung plus "| " sonst.

Die eigentliche Methode **`printTree`** ruft nur unsere *Hilfsfunktion* auf (mit dem Wurzelknoten, mit "darüber", und mit leerer Einrückung).

Im *Hauptprogramm* rufen wir **`printTree`** nach der normalen Baum-Ausgabe auf.

Zusatzaufgabe 2: Statistiken

Wir wollen wissen, wie groß und wie "gut" unser Baum ist.

- Für die *Größe* (= *Anzahl der Knoten im Baum*) bekommt unsere Tree-Klasse eine *Methode* **`size`** (*keine* Parameter, Returntyp **`int`**).

  Sie ruft nur eine *rekursive Hilfsfunktion* auf (ebenfalls Returntyp **`int`**), die die *Knotenanzahl* des jeweiligen *Teilbaumes* zurückliefert.

- Weiters bekommt unsere **`Tree`**-Klasse eine *Methode* **`statistics`** (*ohne* Parameter und *ohne* Returnwert), die umfangreiche *Tiefen-Statistiken* zum Baum *berechnet und ausgibt*.

  - Zuerst einmal berechnet sie die *Knotenanzahl* durch Aufruf der *Größen-Hilfsfunktion* aus dem vorigen Punkt.
  - Dann berechnet und speichert sie die *maximale Tiefe* des Baumes. Schreib dafür eine *rekursive Hilfsfunktion* ähnlich der Größen-Hilfsfunktion.

    In diesem Beispiel möge folgende *Definition der Tiefe* gelten: Ein *leerer* Baum hat Tiefe 0, ein Baum mit *einem* Knoten hat Tiefe 1 usw. (d.h. die *maximale Tiefe* entspricht der *Anzahl der Ebenen* mit Knoten, bzw. die Ebenen werden beginnend mit 1 nummeriert).

    Tipp für die Hilfsfunktion: Der Header **`algorithm`** enthält eine Funktion **`max`**.
  - Als nächstes wird ein *Array* mit einem **`int`** pro Baum-Ebene *dynamisch angelegt* (und am Ende der Methode wieder *freigegeben*!) und *komplett auf 0* gesetzt.

    Dann wird eine *weitere rekursive Hilfsfunktion* aufgerufen, um dieses Array mit der *Anzahl der Knoten pro Ebene* zu befüllen.

    Diese Hilfsfunktion hat außer dem aktuellen Knoten *zwei weitere Parameter*: Das *Array* und den *Index im Array*, der dem aktuellen Knoten entspricht (= Index des Array-Elementes, das *für den aktuellen Knoten um 1 erhöht* wird).
  - Unter Verwendung dieses Arrays wird eine *Liste mit der Verteilung der Knoten* ausgegeben: Jede Zeile soll die *Tiefe*, die *Anzahl der Knoten* auf dieser Ebene, und den *prozentuellen Anteil* der Ebene an der Gesamt-Knotenzahl (als Kommazahl!) enthalten.
    Achtung: Der Array-Index beginnt bei 0, unsere Ebenen-Nummern bei 1 !
  - Als nächstes berechnen wir die *mittlere Tiefe* der Knoten unseres Baumes (diese entspricht dem *mittleren Such-Aufwand*, um einen Knoten zu finden).

    Dazu *summieren* wir in einer *Schleife* über unser Array *"Knotenanzahl auf dieser Ebene mal Tiefe der Ebene"* (Achtung: Tiefe ist wieder Array-Index + 1). Diese Summe wird nach der Schleife durch die Knoten-Anzahl *dividiert* (exakte Komma-Division!). Achte bei der Division auf den *leeren* Baum

(ein leerer Baum hat klarerweise mittlere Tiefe 0).

- Zum Vergleich berechnen wir die *mittlere Tiefe eines optimalen Baumes* mit *gleichvielen* Knoten. Obwohl das keine rekursive Funktion ist, lagern wir die Berechung zur Übersichtlichkeit in eine *eigene Hilfsfunktion* aus. Diese hat einen **`int`**-*Parameter* (*Gesamtzahl* der Knoten) und liefert einen **`double`**-*Returnwert* (*mittlere Tiefe* des optimalen Baumes mit so vielen Knoten).

  Für 0 Knoten liefert die Funktion sofort 0.

  Sonst *summieren* wir mit einer *Schleife* wieder *"Knotenanzahl auf dieser Ebene mal Tiefe der Ebene"*, wobei die Anzahl der Knoten pro Ebene eine *aufsteigende Zweierpotenz* ist (1, 2, 4, 8, ...). Bei jedem Durchlauf ziehen wir diese Knotenanzahl von der Gesamtzahl ab, und die Schleife endet, wenn die *verbleibende Knotenanzahl kleiner als die nächste Zweierpotenz* ist.

  Nach der Schleife müssen wir noch *"verbleibende Knotenanzahl mal Tiefe der untersten Ebene"* zur Summe dazuzählen und die Summe durch die Knoten-Gesamtzahl dividieren (als Komma-Division).

- Am Ende *gibt* unsere Statistik-Methode *die 3 Tiefen aus*: Maximale Tiefe, mittlere Tiefe, mittlere Tiefe bei optimalem Baum.

Unser *Hauptprogramm* soll gleich nach der Liste aller Worte die *Größe* des Baumes (d.h. die Anzahl verschiedener Worte in der Eingabe) ausgeben und ganz am Ende die *Statistik*-Methode aufrufen.

Zusatzaufgabe 3: `getFirst` und `getNext`

Derzeit kann nur unsere **`Tree`**-Klasse intern einen Baum *Knoten für Knoten durchlaufen*, nicht aber z.B. unser Hauptprogramm.

Deshalb erweitern wir die Klasse **`Tree`** um *zwei Methoden*, die ein knotenweises Durchlaufen des Baumes in Sortier-Reihenfolge ermöglichen. Beide liefern einen *Pointer auf einen einzelnen Baumknoten als Returnwert*:

- **`getFirst`** soll einen Pointer auf den *sortierordnungsmäßig kleinsten Knoten* im Baum liefern. Diese Methode hat *keinen* Parameter.

  Ist der Baum *leer*, returniert sie sofort den *Nullpointer*.

  Sonst geht sie im Baum mit einer *Schleife* so lange *immer wieder nach links*, bis sie bei einem Knoten ist, der *keinen linken Sohn* mehr hat. Dieser Knoten ist das Ergebnis.

- **`getNext`** wird mit einem **`string`**-*Parameter* aufgerufen und soll einen Pointer auf den ersten (kleinsten) Knoten liefern, der *sortierordnungsmäßig größer* als das angegebene Wort ist (oder den *Nullpointer*, falls es *keinen* solchen Knoten gibt).

  Dazu *durchwandert* die Methode den Baum oben beginnend mit einer *Schleife* so ähnlich wie beim Suchen, bis es nicht mehr weiter geht, und *merkt* sich auf dem Weg denjenigen Knoten, der am nächsten beim gesuchten Wort liegt.

  - Ist der Pointer auf den aktuellen Knoten *Null*, *returniert* die Methode den *zuletzt gespeicherten Pointer* (womit *initialisierst* du den zuletzt gespeicherten Pointer *vor* der Schleife?).

- Ist der aktuelle Knoten *größer als das angegebene Wort*,
  wird er als potentielles Ergebnis *gespeichert*,
  und die Schleife geht in seinen *linken* Unterbaum.
- Sonst (d.h. wenn der aktuelle Knoten *kleinergleich* dem angegebenen Wort ist)
  wird der aktuelle Knoten *ignoriert*, und die Schleife macht *rechts* davon weiter.

Damit unser Hauptprogramm mit den **Node**-Pointern, die es von **`getFirst`** und **`getNext`** bekommt, auch etwas Sinnvolles machen kann, geben wir unserer **Node**-*Klasse zwei öffentliche Get-Methoden* für das *Wort* und die *Anzahl* des Wortes.

Im *Hauptprogramm* ersetzen wir die Ausgabe des Baumes mittels Ausgabe-Operator durch eine *Schleife*. Diese Schleife sieht so ähnlich aus *wie eine Listen-Durchlauf-Schleife*:

- Sie hat einen **Node**-*Pointer* als Schleifen-Variable.
- Dieser wird mit **`getFirst`** initialisiert.
- Die Schleife läuft, solange der Pointer *nicht Null* ist.
- Am *Ende* jedes Durchlaufes rückt man den Pointer weiter,
  indem man **`getNext`** mit dem *Wort des aktuellen Knotens* aufruft.

In der Schleife geben wir den Knoten, auf den der Pointer gerade zeigt,
mit dem *einfachen Knoten-Ausgabe-Operator* aus.

Zusatzaufgabe 4: Knoten löschen

Bevor wir uns dem Löschen widmen, wollen wir den bisher als "verboten" markierten *Copy-Konstruktor* in unserer **`Tree`**-Klasse implementieren. Dazu sind 3 Schritte nötig:

- Unsere **Node**-Klasse bekommt einen *zweiten Konstruktor*,
  bei dem *alle vier Member-Werte als Parameter* angegeben werden.
- Die **`Tree`**-Klasse bekommt eine *rekursive Hilfsfunktion zum Kopieren von Bäumen*.
  Sie hat wie üblich einen *Pointer auf den zu kopierenden Teilbaum als Parameter*
  und liefert einen *Pointer auf den obersten Knoten der Kopie als Returnwert*.

  Wird sie für einen *leeren Teilbaum* aufgerufen, liefert sie einen *Nullpointer* zurück.
  Sonst gibt sie einen *neu angelegten Knoten* mit *denselben Nutzdaten wie im Original*
  zurück, an dem links und rechts *Kopien der Unterbäume* des Originals hängen.
- Der *Copy-Konstruktor* der **`Tree`**-Klasse initialisierst die Wurzel des neuen Baumes
  mit einer *rekursiven Kopie der Wurzel des Original-Baumes*.

Dann bekommt unsere **`Tree`**-Klasse eine *Lösch-Methode* **`remove`**.
Sie hat einen **`string`**-*Parameter*: Das *Wort*,
dessen Knoten aus dem Baum entfernt werden soll.
Als *Returnwert* liefert sie einen **`int`**: Den *Zählerstand* des gelöschten Wortes oder 0,
wenn das angegebene Wort im Baum gar nicht vorkommt.

Für die Implementierung von **`remove`** gibt es *mehrere Varianten*:

- Die *"ganz normale"* mit Schleife und vielen Fallunterscheidungen
  (lösche ich die Wurzel, einen linken Sohn oder einen rechten Sohn?).
- Die *"fortgeschrittene"* Version der ersten Variante, die sich
  einen Teil der Fallunterscheidungen mittels *Pointer-auf-Pointer-Trick* spart
  und mit weniger Code auskommt.

- Eine *rekursive Variante*, die zwar etwas ineffizienter ist,
  aber vielleicht am einfachsten verständlich.

Zur "normalen" Variante:

- Wir brauchen *drei Variablen*:
  - Einen *Pointer*, der später *auf den zu löschenden Knoten* zeigen wird
    (wo *beginnen* wir mit diesem Pointer?).
  - Einen *Pointer*, der *auf dessen Vater* zeigt (am *Anfang* ist das der **`nullptr`**).
  - Einen **`bool`**-Wert, der uns sagt, ob wir *beim letzten Schritt* im Baum
    zum linken oder zum rechten Sohn abgestiegen sind.
- Dann *suchen* wir unser Wort wie üblich mit einer *Schleife* im Baum:
  - Landen wir bei einem **`nullptr`**, geben wir **`0`** zurück.
  - *Finden* wir unser Wort, *verlassen* wir die Schleife.
  - Steigen wir *links oder rechts* ab, müssen wir die drei oben genannten *Variablen*
    alle entsprechend neu setzen.
- Wir *merken* uns den *Zählerstand* des aktuellen Knotens als späteren *Returnwert*.
- Wenn der aktuelle Knoten *zwei nichtleere Unterbäume* hat,
  haben wir den *komplizierten Fall* vor uns: Wir müssen
  den *größten Knoten des linken Unterbaumes* suchen und heraufkopieren.

  Also brauchen wir *zwei weitere Pointer*:
  Einen müssen wir auf *den heraufzukopierenden Knoten* zeigen lassen,
  den anderen auf *dessen Vater*. Dazu müssen wir vom zu löschenden Knoten
  ein Mal nach links gehen und dann *immer wieder nach rechts*,
  bis wir bei einem Knoten *ohne rechten Sohn* sind,
  und uns dabei auch immer dessen Vater merken.

  Dann *kopieren wir das Wort und den Zählerstand* aus dem so gefundenen Knoten
  in den ursprünglich zu löschenden Knoten.

  Als nächstes müssen wir den hinaufkopierten Knoten *aus dem Baum entfernen*,
  indem wir den Zeiger auf diesen Knoten *durch seinen linken Sohn ersetzen*
  (oder durch den **`nullptr`**, wenn er keinen linken Sohn hat):
  - Ist er unmittelbar der *linke Sohn des ursprünglich zu löschenden Knotens*,
    so wird sein linker Sohn der neue linke Sohn des ursprünglichen Knotens.
  - Sonst wird *sein linker Sohn als neuer rechter Sohn seines Vaters* gespeichert.

  Zuletzt wird das hinaufkopierte *Knoten-Objekt gelöscht*.
- Im *einfachen Fall* müssen wir zuerst den Pointer ermitteln,
  der statt dem Pointer auf den zu löschenden Knoten zu speichern ist:
  Wenn der zu löschende Knoten *einen Sohn* hat, ist das der *Pointer auf diesen Sohn*,
  sonst der **`nullptr`**.

  Dann muss dieser Pointer *an der richtigen Stelle gespeichert* werden:
  - Ist der *Vater-Pointer* der **`nullptr`**, so löschen wir gerade die Wurzel.
    Unser Sohn wird daher die *neue Wurzel*.
  - Sind wir beim *letzten Abstieg nach links* gegangen,
    wird unser Sohn der *linke Sohn* unseres Vaters.
  - *Sonst* wird er der *rechte Sohn* unseres Vaters.

  Danach wird das zu löschende *Knoten-Objekt freigegeben*.

- Am Ende wird der *zwischengespeicherte Returnwert* zurückgegeben.

Die Alternative dazu ist die **Pointer-auf-Pointer-Variante** von **`remove`**:

- Wir deklarieren einen *Pointer* auf einen **`Node`**-*Pointer* (ich nenne ihn hier **`PtrPtr`**) und lassen ihn am Anfang auf das *Wurzel-Pointer-Member* unseres Baumes zeigen. Dafür fallen der Vater-Pointer und das **`bool`**-Flag weg.
- Dann kommt die *Such-Schleife*:
  - Der *aktuelle Pointer* (Pointer auf den aktuellen Knoten) ist der Pointer, auf den **`PtrPtr`** gerade zeigt.
  - Ist der aktuelle Pointer *gleich Null*, geben wir sofort 0 zurück (*"nicht gefunden"*).
  - Ist das Wort im aktuellen Knoten *kleiner* als das zu löschende Wort, lassen wir **`PtrPtr`** auf den *rechten Sohn-Pointer im aktuellen Knoten* zeigen.
  - Ist das Wort im aktuellen Knoten *größer* als das zu löschende Wort, lassen wir **`PtrPtr`** auf den *linken* Sohn-Pointer im aktuellen Knoten zeigen.
  - Sonst haben wir den Knoten mit dem gesuchten Wort *gefunden* und *beenden* die Schleife, um ihn zu löschen.
- Wir *merken* uns wieder den *Zählerstand* des aktuellen Knotens für den Returnwert.
- Wenn der aktuelle Knoten *zwei nichtleere Unterbäume* hat, haben wir den *komplizierten Fall* vor uns: Er ist ident zur *normalen Lösung*.
- Sonst müssen wir dort, wo **`PtrPtr`** hinzeigt, den *Ersatz für den aktuellen Knoten* speichern: Ist der *linke Sohn* des aktuellen Knotens *nicht Null*, so ersetzt er den aktuellen Knoten, sonst der *rechte Sohn*.

  Danach wird wie bisher das aktuelle *Knoten-Objekt freigegeben*.

Zur **rekursiven Variante**:
Wie üblich ruft die eigentliche Methode dafür nur eine *rekursive Hilfsfunktion* auf.

- Diese rekursive Hilfsfunktion hat *zwei Parameter*: Das zu löschende *Wort* und den *Pointer auf den aktuellen Teilbaum*. Der wesentliche Trick ist, dass dieser Pointer-Parameter *"by Reference"* übergeben werden muss: Wenn die rekursive Hilfsfunktion den *obersten Knoten* ihres Teilbaumes *löscht*, muss sie den *Pointer im Aufrufer ändern*. Sie lässt ihn auf den *neuen obersten Knoten* des Teilbaumes zeigen (oder setzt ihn *auf Null*, wenn der gelöschte Knoten der einzige Knoten im Teilbaum war).
- Als erstes kommen die *drei einfachen Fälle*:
  - Ist der aktuelle Teilbaum *leer*, liefert die Funktion 0 zurück (*"nicht gefunden"*).
  - Ist das Wort im aktuellen Teilbaum *kleiner* als das zu löschende Wort, wird im *rechten* Unterbaum weitergelöscht, ist es *größer*, dann *links*.
- Sonst hat man den *Knoten mit dem zu löschenden Wort* gefunden und muss ihn entfernen.
  Wir merken uns zuerst einmal seinen *Zählerstand* für den *Returnwert*.
  Dann müssen wir 3 Fälle unterscheiden:
  - Der *linke* Sohn-Pointer ist *Null*.
    Dann *merken* wir uns den *rechten Sohn*, *löschen* das aktuelle Knoten-*Objekt*, und *ändern* den aktuellen Pointer *auf den bisherigen rechten Sohn*.

    Das deckt auch den Fall ab, dass *beide Söhne Null* sind:

in diesem Fall wird richtigerweise Null im aktuellen Pointer gespeichert.

- Der *rechte* Sohn-Pointer ist *Null*. Dann passiert das Umgekehrte:
  Linken Sohn merken, aktuelles Knoten-Objekt löschen,
  aktuellen Pointer auf den gemerkten linken Sohn zeigen lassen.
- Sonst haben wir den *komplizierten Fall*: Der zu löschende Knoten
  hat *zwei Söhne* und kann daher *nicht direkt gelöscht* werden.

  Für diesen Fall brauchen wir zuerst einmal eine kleine *Hilfsfunktion*,
  die mit einem Knoten-Pointer aufgerufen wird und einen Pointer auf den
  größten (am weitesten rechts stehenden) Knoten in diesem Teilbaum liefert.
  Diese Funktion geht spiegelbildlich zu **`getFirst`** mit einer *Schleife* einfach
  so lange nach rechts, bis sie bei einem *Knoten ohne rechten Sohn* angelangt ist.

  Mit dieser Hilfsfunktion suchen wir den *größten Knoten des linken Unterbaumes*
  und *kopieren* dessen *Wort* und dessen *Anzahl* in den *aktuellen Knoten*.

  Dann rufen wir unsere Lösch-Hilfsfunktion *rekursiv* mit dem soeben kopierten
  Wort für unseren *linken Unterbaum* auf, um den heraufkopierten Knoten dort
  zu löschen.

In allen drei Fällen geben wir dann den *gemerkten Wert zurück*.

Unser Hauptprogramm soll neben dem Baum mit allen Wörtern
auch einen Baum mit *allen mehrfach vorkommenden Wörtern* bekommen.
Dazu ändern wir es wie folgt:

- Nach dem Einlesen der Eingabe legen wir ein *zweites lokales Tree-Objekt*
  als *Kopie des ersten* an.
- Dann gehen wir den *zweiten Baum* mit einer **`getFirst`**/**`getNext`**-*Schleife*
  Knoten für Knoten durch.

  In der Schleife *merken* wir uns zuerst einmal das *Wort* des aktuellen Knotens
  für das nächste **`getNext`** (sonst könnten wir ja nach dem Löschen des Knotens
  kein **`getNext`** mehr machen).

  Dann prüfen wir, ob der aktuelle Knoten *Zählerstand 1* hat.
  Wenn ja, *löschen* wir ihn mit unserem **`remove`**.
  Zur Sicherheit prüfen wir noch, ob auch das **`remove`** 1 als Returnwert liefert.
- Am Ende geben wir *beide Bäume* nacheinander mittels *Ausgabe-Operator* aus.

Alternative Lösung mit STL `map`-Template

Abschließend wollen wir ein Programm mit derselben Funktionalität
unter Verwendung des *vordefinierten Templates* **`map`** implementieren.
Benutze für die Details die *Online-Doku* von **`map`**, **`pair`** usw.!

- Unsere *beiden eigenen Klassen* sowie die *drei Ausgabe-Operatoren*
  fallen *komplett weg*.
- Dafür kommen *zwei neue Ausgabe-Operatoren* dazu.
  Schreib beide *als Template* mit zwei Typ-Parametern,
  d.h. so, dass sie für beliebige **`map`**-Objekte verwendbar sind.
  - Der erste wird mit einem *einzelnen Element* der **`map`** aufgerufen,

d.h. mit einem **`pair`**.
Er gibt die beiden Member des **`pair`** durch **`": "`** getrennt aus.

- Der zweite wird mit einer **`map`** aufgerufen.
  Er *durchläuft* die **`map`** mit einer *Schleife* und gibt die einzelnen Elemente unter Verwendung des ersten Ausgabe-Operators aus.

  Probiere beide *Schleifen-Varianten*:
  Die klassische mit einem *Iterator als Schleifen-Variable*
  und die moderne mit **`for ( … : … )`**.

- Die **`getword`**-Funktion bleibt völlig unverändert.
- Das **`main`** ändert sich wie folgt:
  - Unsere beiden Bäume sind jetzt vom Typ **`map<string, int>`** .
  - In der Lese-**`while`**-Schleife *suchen* wir das eingelesene Wort mit **`find`** in der **`map`**. *Finden* wir es (**`find`** liefert einen *Iterator*, der darauf zeigt), so *erhöhen* wir seinen Zähler um 1.

    Finden wir es *nicht*, fügen wir mit **`insert`** ein *frisch erzeugtes Pärchen* mit dem gelesenen Wort und Zählerstand 1 in die **`map`** ein
    (dazu brauchst du einen **`pair`**-Konstruktor-Aufruf).

    Alternativ kann man *beide Fälle* mit einer *einzigen Code-Zeile* abdecken:
    Der **`[]`**-Operator kann sowohl *bestehende Elemente* einer **`map`** *suchen* als auch *neue einfügen und initialisieren*.

  - Die *Lösch-Schleife* ersetzen wir durch eine *Iterator-Schleife* über die **`map`**.
    In der Schleife prüfen wir, ob der *Zählerstand* (zweites Member) im aktuellen Element 1 ist. Wenn ja, *löschen* wir es mit **`erase`**.

    Achtung: Es ist *nicht* klug, für einen Iterator *"Weiterzählen"* ("geh auf das nächste Element") aufzurufen,
    nachdem das Element, auf das der Iterator zeigt, *gelöscht* wurde:
    Nach dem Löschen des betreffenden Elementes ist der Iterator *ungültig*.

    Du solltest daher den Schleifen-Iterator zuerst in eine *Hilfsvariable* kopieren, dann den Schleifen-Iterator *weiterzählen*, und *erst danach* das Element prüfen und eventuell löschen, auf das die Hilfsvariable zeigt.