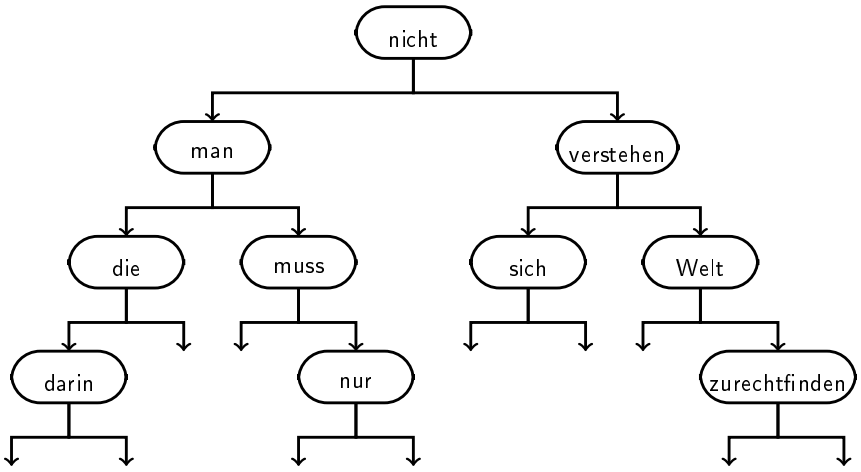
9 Bäume

Bäume funktionieren ähnlich wie verkettete Listen, sie werden durch **Referenzen** zusammengehalten. Der grosse Unterschied besteht darin, dass **Baumelemente** mehr als eine Referenz erlauben. Wir wollen uns im Folgenden auf einen **Binärbaum** beschränken, das heisst jeder Knoten bietet genau **zwei Referenzen** (Links) an. Dadurch wird aus der eindimensionalen verketteten Liste ein zweidimensionales Gebilde, eben ein Baum.



Bäume stellt man sich in der Regel als nach unten wachsend vor.

Üblicherweise werden die `null`-**Referenzen** an den Enden nicht dargestellt, aber alle Knoten verfügen bei einem Binärbaum über genau zwei Referenzen. Elemente, bei denen beide Referenzen je `null` sind, werden als Blätter bezeichnet.

Beispielsweise wollen wir nun das das Element "*sich*" suchen. Für jede Suche starten wir bei der Wurzel des Baumes. Die `Strings` sind lexikografisch geordnet. "*sich*" ist daher grösser als "*nicht*", deswegen gehen wir nach rechts. Wir erreichen "*verstehen*". "*sich*" ist kleiner als "*verstehen*", deswegen gehen wir nun nach links. Und schon haben wir "*sich*" gefunden.

Angenommen wir hätten eine Datenstruktur mit tausend Elementen. In einer verketteten Liste, müssten wir durchschnittlich 500 Element durchgehen, um ein beliebiges Element zu finden. In einem Baum können wir bei jedem Knoten einen Vergleich anstellen und wissen anschliessend, in welche Richtung wir weiter müssen. Die Tiefe des Baume bestimmt die maximale Suchdauer. Bei einem schön verteilten (balancierten) Binärbaum ist die Tiefe nur 10 ($\lceil log_2(1000) \rceil = 10$), ein Baum ist in diesem Beispiel also $50\times$ schneller, als eine Liste.

Beim Einfügen gehen wir gleich vor, wie beim Suchen. Wenn das einzufügende Element (wie erwartet) noch nicht Teil des Baumes ist, landen wir schlussendlich bei einer `null`-Referenz. Genau dort hängen wir dann das neue Element an. Duplikate von Strings sind in einem Baum nicht vorgesehen. Wenn wir ein Wort beim Versuch des Einfügens bereits vorfinden, brechen wir das Einfügen einfach ab.

9.1 Binärbaum

a) Erstellen Sie in Eclipse ein neues Projekt "Tree".

b) Verwenden Sie intern eine Hilfsklasse `TreeElement`. Die Klasse `TreeElement` muss über zwei **reflexive Referenzen** `TreeElement lower` und `TreeElement higher` verfügen. Als Nutzdaten soll `TreeElement` wie im Bild einen **String** enthalten. Da Strings mittels der Methode `compareTo` sortierbar sind, dient dies als Kriterium wann dem Pfad `lower` und wann `higher` gefolgt werden soll.

c) Programmieren Sie eine Klasse `BinaryTree`, die mittels `TreeElement` einen Binärbaum implementiert. Die Klasse `BinaryTree` muss nur die Referenz zum **Wurzelelement** speichern.

d) Implementieren Sie die Methoden `void add(String s)` und `String toString()` in `BinaryTree`. Die Methode `String toString()` soll den ganzen Baum in alphabetischer Reihenfolge und durch Leerzeichen getrennt zurückgeben. Traversieren Sie den Baum entsprechend.

e) Fügen Sie eine Klasse `Demo` zum Projekt hinzu, die die `main`-Methode enthält.

f) Vielleicht haben Sie es gemerkt, die Wörter aus der Grafik oben, stammen aus einem Zitat von Albert Einstein. Hier ist ein weiteres Zitat von Einstein: "Zwei Dinge sind unendlich, das Universum und die menschliche Dummheit, aber bei dem Universum bin ich mir noch nicht ganz sicher." Fügen Sie die Wörter des Zitats mittels add in ein Objekt von Ihrem `BinaryTree` ein und geben Sie den Baum anschliessend mit `System.out.println` auf der Kommandozeile aus. Was erhalten Sie?

g) Implementieren Sie eine dritte Methode `int depth()` in `BinaryTree`. Erfinden Sie dazu einen Algorithmus, der die maximale Tiefe des Baumes bestimmt. Welche Tiefe erhalten Sie für das Einstein-Zitat?

h) Überprüfen Sie nun meine Behauptung zur Suchtiefe in einem Baum. Fügen Sie tausend Zufallszahlen in einen Baum ein. Verwenden Sie `Integer.toString(int i)`, um die Zufallszahlen in einen `String` zu konvertieren. Welche Tiefe erhalten Sie?

i) Wie sieht es hingegen aus, wenn Sie stattdessen die Zahlen von 100 bis 999 sortiert einfügen? Was stellen Sie fest? Begründen Sie Ihre Erkenntnis.

9.2 Persistenter Highscore

Ziel dieser Aufgabe ist, dass eine Menge erreichter Scores in eine Datei geschrieben und wieder von der Datei gelesen werden können. Im folgenden geben wir detaillierte Anleitung, um zu diesem Ziel zu gelangen. Bereits vor vielen Wochen haben wir dieses Kapitel versprochen. Das Problem ist aber, dass Dateien bis jetzt noch nicht Teil der Vorlesung sind. Es gibt aber auch nicht viel Theoretisches zum Arbeiten mit Dateien zu sagen. Nichtsdestotrotz halten wir es für wichtig im Programmieralltag.

a) Nehmen Sie die Klassen Highscore (Übung 6.1) und Player (Übung 3.4) zur Hand. Falls Sie nicht Ihren Code verwenden wollen, finden Sie Beispiellösungen auf der Vorlesungswebsite.

b) Wir wollen nun eine Subklasse von Highscore ableiten, die das Speichern in und Lesen aus einer Datei unterstützt. Stellen Sie deswegen sicher, dass das Feld in `Highscore` nur `protected` ist. Ansonsten haben wir in der Subklasse keinen Zugriff auf das Feld.
Richtig: `protected LinkedList<Player> highscore;`
Falsch: `private LinkedList<Player> highscore;`

c) Erstellen Sie nun eine neue öffentliche Klasse z. B. "`FiledHighscore`", die von `Highscore` ableitet. Wir machen das in Vorraussicht auf Datenbanken. Weil analog zu Dateien, können wir unsere Highscores dann auch in einer Datenbank ablegen. Wenn Sie Eclipse beim Erstellen der neuen Klasse `FiledHighscore` die Superklasse `Highscore` gerade mitteilen, kann Eclipse den entsprechenden Konstruktor für Sie erstellen. Setzen Sie dazu den Haken bei "Constructors from Superclass".

d) Importieren Sie die nötigen Java-Klassen zum Arbeiten mit Dateien. Fügen Sie dazu "import java.io.*;" zu Ihrem Programmkopf hinzu. Das "io" steht für Input/Output.

e) Erstellen Sie zwei leere Methoden "`store`" und "`load`".

f) Beim Arbeiten mit Dateien kann so einiges schief gehen. Zum Beispiel können die Dateien gleichzeitig von anderen Programmen verwendet werden oder schreibgeschützt sein. Deswegen müssen wir in unsere Methoden eine entsprechende Fehlerbehandlung einbauen. Wenn im `try`-Block ein Fehler Auftritt kann er im `catch`-Block entsprechend behandelt werden. Der untenstehende Code reicht vollauf als Grundgerüst für den Code in `store` und `load`. `IOExceptions` werden gefangen und ausgegeben.

```
try {
    // TODO: Add method code
} catch (IOException e) {
    e.printStackTrace();
}
```

g) Unter Verwendung der folgenden Klassen erhalten Sie schnell und einfach Schreibzugriff auf die Datei "`highscore.txt`":
`BufferedWriter out = new BufferedWriter(new FileWriter("highscore.txt"));`
Wenn die Datei noch nicht existiert, wird sie automatisch neu erstellt.

h) Mittels `out.write(...)` können Sie zuerst `score` und dann `name` von jedem `Player` in die Datei schreiben. Verwenden Sie für jeden `Player` eine Zeile in der Datei und schliessen Sie diese mit einem Zeilenumbruch (`'\n'`) ab.

i) Mit der Methode `close()` schliessen Sie den jeweiligen Stream zum Schluss. Ansonsten meldet sich Eclipse mit einer Fehlermeldung.

j) Schreiben Sie in `main` Testwerte in ein Objekt von Ihren `FiledHighscore` und testen Sie Ihre `store`-Methode. Sie können die resultierende Datei mit einem Texteditor anschauen. Die gespeicherte `int`-Zahl resultiert in einem einzigen (Sonder-)Zeichen.

k) Zum Lesen der Datei verwenden wir folgende Klassen:
`BufferedReader in = new BufferedReader(new FileReader("highscore.txt"));`

l) Mit `in.read()` lesen Sie zuerst den Score und anschliessend mit `in.readLine()` den Namen des Players. Mit `in.ready()` können Sie jeweils abfragen, ob die Datei noch mehr Daten zu bieten hat.

9.3 Weiterführendes Grafikprojekt

Freiwillig: Falls Sie Lust haben, mit der Klasse `EasyGraphics` lassen sich viele interessante Spielereien realisieren. Hier nur ein paar Ideen:

- Farbige Muster zeichnen (Linien, Spiralen, ...)
- Baum visualisieren (Nur die Äste, keine Strings)
- Pixel, Rechteck oder Kreis durchs Bild fliegen und von den Rändern abprallen lassen
- Game of Life: Zufallsmuster vorgeben & starten: Leere Pixel mit genau drei Nachbarn erwachen zum Leben. Pixel mit weniger als zwei Nachbarn oder mehr als drei Nachbarn sterben.
- Pixelwurm durch den Bildschirm fahren lassen
- Kanone, die Pixel verschiesst

Für Animationen müssen Sie nach dem `repaint` jeweils eine Wartezeit einbauen.

```
try {
    Thread.sleep(20); // 20ms (e.g.)
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Da zum Beenden des Programmes oben am Fenster ein Button zur Verfügung steht, können Sie für Animationen in der `main`-Methode mit einer Endlosschleife arbeiten.

```
for (;;) { // Animation
    // a) Set pixels
    // b) Repaint
    // c) Sleep
    // d) Reset pixels
}
```

Beim Abfragen von Maus und Tasten helfen wir auf Anfrage gerne weiter.

Für das beste Projekt vergeben wir nach Wahl eine Flasche Champagner oder einen grossen Osterhasen.

Projektabgaben bitte an den jeweiligen Assistenten und CC an `felix.friedrich@inf.ethz.ch` und `lars.widmer@inf.ethz.ch`.

Viel Glück & Erfolg!