Theorie

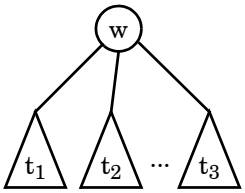
1. Grundbegriffe

Bäume sind verallgemeinerte Listenstrukturen. Ein Element – üblicherweise spricht man von Knoten (*node*) – hat nicht, wie im Falle linearer Listen, nur einen Nachfolger, sondern eine endliche, begrenzte Anzahl von Söhnen. In der Regel ist einer der Knoten als Wurzel (*root*) des Baumes ausgeprägt. Das ist zugleich der einzige Knoten ohne Vorgänger. Jeder andere Knoten hat einen (unmittelbaren) Vorgänger, der auch Vater des Knotens genannt wird. Eine Folge p_0, …, p_k von Knoten eines Baumes, welche die Bedingung erfüllt, dass p_{i+1} Sohn von p_i ist für $0 \le i < k$, heisst Pfad (*path*) mit Länge k, der p_0 mit p_k verbindet. Jeder von der Wurzel verschiedene Knoten eines Baumes ist durch genau einen Pfad mit der Wurzel verbunden. Da die Menge der Knoten eines Baumes stets als endlich vorausgesetzt wird, muss es Knoten geben, die keine Söhne haben. Diese Knoten werden als Blätter (*leaf*) bezeichnet; alle andern Knoten nennt man innere Knoten.

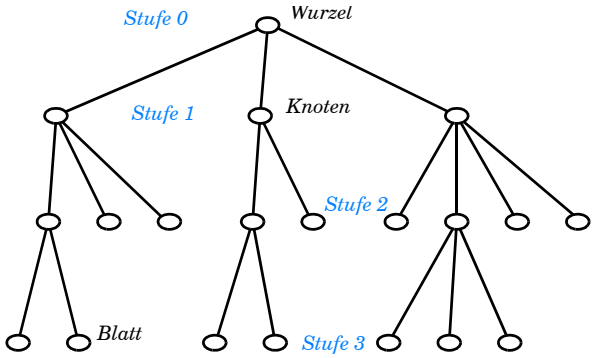
Ist unter den Söhnen eines jeden Knotens eines Baumes eine Anordnung definiert, so dass man vom ersten, zweite, dritten usw. Sohn des Knotens sprechen kann, so nennt man den Baum geordnet. Dies darf man jedoch nicht mit der Ordnung eines Baumes (Grad) verwechseln, welches die maximale Anzahl von Söhnen eines Knotens ist. Besonders wichtig sind geordnete Bäume der Ordnung 2 (*binäre Bäume* oder Binärbäume, *binary trees*). Statt vom ersten und zweiten Knoten spricht man dann vom linken und rechten Sohn eines Knotens.

Die Menge aller Bäume der Ordnung d, kann man auch rekursiv definieren:

- Der aus einem einzigen Knoten bestehende Baum ist ein Baum der Ordnung d.
- Sind t_1, …, t_d beliebige Bäume der Ordnung d, so erhält man einen (weiteren) Baum der Ordnung d, indem die Wurzeln von t_1, …, t_d zu Söhnen einer neu geschaffenen Wurzel w macht. t_i $(1 = i = d)$ heisst i-ter Teilbaum der Wurzel w.



Bäume der Ordnung $d > 2$ nennt man auch *Vielwegbäume*. In der Informatik wachsen die Bäume in anderer Richtung als in der Natur: die Wurzel oben, die Blätter unten!



Die Tiefe eines Knotens ist sein Abstand zur Wurzel, d.h. die Anzahl der Kanten auf dem Pfad von diesem Knoten zur Wurzel. Die Höhe eines Baumes ist der maximale Abstand eines Blattes von der Wurzel.

Man fasst die Knoten eines Baumes gleicher Tiefe zu Niveaus (Stufen) zusammen.

Ein Baum heisst vollständig, wenn er auf jedem Niveau die maximal mögliche Knotenzahl hat und sämtliche Blätter dieselbe Tiefe haben.

2. Binäre Suchbäume

Ein binärer Suchbaum ist ein geordneter Baum mit Ordnung $d = 2$. In jedem Knoten wird ein Suchschlüssel so abgespeichert, dass alle Suchschlüssel des linken Teilbaums des Knotens kleiner und alle Suchschlüssel des rechten Teilbaums des Knotens grösser sind. Das heisst, dass an jedem Knoten alle kleineren Suchschlüssel über den linken Sohn und alle grösseren Suchschlüssel über den rechten Sohn erreicht werden.

Die unteren randständigen Knoten sind üblicherweise mit Null-Referenzen abgeschlossen, wodurch das Ende eines Baumes bei dessen Traversierung einfach erkannt werden kann.

Im folgenden werden wichtige Grundoperationen auf binären Suchbäumen besprochen:

2.1 Suchen

Aufgrund der Ordungsbedinung in einem Suchbaum (alle Knoten im linken Teilast unter dem Knoten n besitzen kleinere, alle Knoten im rechten Teilast unter dem Knoten n grössere Schlüssel als n) verläuft die Suche nach einem bestimmten Schlüssel nach einem einfachen iterativen Muster:

1. Beginne bei der Wurzel des Baumes. Mache ihn zum aktuellen Knoten `current`.
2. Falls `current.key` der gesuchte Schlüssel ist, breche mit Erfolg ab. Ist der gesuchte Schlüssel kleiner, mache `current.L` zum aktuellen Knoten `current`. Ist der gesuchte Schlüssel grösser, mache `current.R` zum aktuellen Knoten `current`.
3. Ist `current` gleich `null`, breche mit Misserfolg ab, ansonsten geht es weiter mit Schritt 2.

Mit jeder Iteration über die Schritte 2 und 3 wandert die Referenz `current` um eine Ebene nach unten. Damit ist klar, dass sich die Suchzeit schlimmstenfalls proportional zur maximalen Baumtiefe verhält. In einem völlig ausgeglichenen Binärbaum beträgt seine Höhe etwa $log_2(N)$. Die obere Grenze für den Suchaufwand ist demzufolge $O(log\ N)$. Durch ungünstige Einfüge-Operationen kann ein Baum entarten, d.h. seine Ausgeglichenheit verlieren. In der schlimmsten Form der Entartung erhält ein Baum die Gestalt einer linearen Liste, und die durchschnittliche Suchzeit wird proportional zu $N/2$.

2.2 Einfügen

Die Operation *Einfügen* beinhaltet zunächst sie Suche nach der geeigneten Einfügestelle, welche in zwei verschiedenen Fällen enden kann:

a) Es existiert bereits ein Knoten mit dem gesuchten Schlüssel.

b) Der Schlüssel wurde nicht gefunden, existiert also im Baum noch nicht.

Im *Fall a)* können wir auf zwei Arten weiterfahren. Da jeder Schlüssel nur genau einmal in Baum existieren darf, können wir entweder das Datenobjekt des gefundenen Knotens durch das neue Datenobjekt ersetzen oder die Operation ignorieren, oder aber einen Fehler werfen.

Im *Fall b)* hat die Suche an einer `null`-Referenz geendet. Genau an dieser `null`-Referenz wird nun ein neuer Knoten angehängt und mit dem Datenobjekt befüllt. Die Referenzen auf Teilbäume des neuen Knotens werden mit `null` initialisiert.

Werden neue Element in ungünstiger Reihenfolge eingefügt, entartet der Baum und büsst seine gute Performance beim Suchen ein.

2.3 Löschen

Beim *Löschen* eines Knotens müssen wir vier Fälle unterscheiden, zunächst muss aber der zu löschende Knoten gesucht werden, wie beim Einfügen. Falls der Knoten gefunden wird, heisse er c:

a) *Knoten existiert nicht.* → Ignorieren oder Fehler rapportieren.

b) *c hat keine Söhne.* → Im Vater-Knoten die Referenz auf c mit `null` überschreiben.

c) *c hat einen Sohn.* → Im Vater-Knoten die Referenz auf c mit der Referenz auf dessen Sohn überschreiben.

d) *c hat zwei Söhne.* → Ersetze c entweder durch den Knoten mit dem grössten Schlüssel aus dem linken Teilbaum oder durch den Knoten mit dem kleinsten Schlüssel aus dem rechten Teilbaum. Der Ersatzknoten muss anschliessend an seiner Original-Position gelöscht werden.

2.4 Traversieren

Da die Struktur eines Baumes *rekursiv* definiert ist (die Teiläste unter einem Knoten sind Bäume), findet man elegante und einfache Algorithmen, um alle Knoten eines Baumes zu durchlaufen (und zu besuchen):

```
void traverse(Node root){
  if (root != null){
    visit(root);        // 1: besuche den aktuellen Knoten
    traverse(root.L);   // 2: traversiere den linken Teilbaum
    traverse(root.R);   // 3: traversiere den rechten Teilbaum
  }
}
```

Die Reihenfolge, mit der die Knoten durchlaufen werden, hängt von der Reihenfolge der Operationen 1 bis 3 ab. Sie können beliebig permutiert werden, so dass wir insgesamt 6 verschiedene Reihenfolgen erhalten. Erfolgt der Besuch als erstes, sprechen wir von einer *Preorder*-Traversierung. Bei der *Inorder*-Traversierung liegt der Besuch in der Mitte, bei der *Postorder*-Traversierung am Schluss.

Die *Inorder*-Traversierungen sind deshalb von besonderer Bedeutung, weil sie einen Suchbaum in auf- bzw. absteigender Sortier-Reihenfolge durchlaufen.

Man sucht allerdings in den sechs beschriebenen rekursiven Traversierungen *eine* bestimmte Reihenfolge vergeblich: Die Traversierung nach Stockwerken (*level order traverse*). Da sie den natürlichen Verbindungen zwischen den Knoten im Baum zuwiderläuft, müssen wir einen anderen (iterativen) Ansatz verwenden:

Es sei `q` eine Warteschlange (*queue*). Dann sieht der Algorithmus für die Level-Order-Traversierung so aus:

```
Node current;
q.push(root);
while (!q.isEmpty()){
  current = q.pop();
  visit(current);
  q.push(current.L);
  q.push(current.R);
}
```

Während der Speicherbedarf (*stack load*) bei der rekursiven Traversierung maximal *log2(N)* beträgt, wächst der Füllstand der Warteschlange kurz vor Beginn der letzten Ebene auf *N/2* an (ausgeglichener Baum vorausgesetzt)!