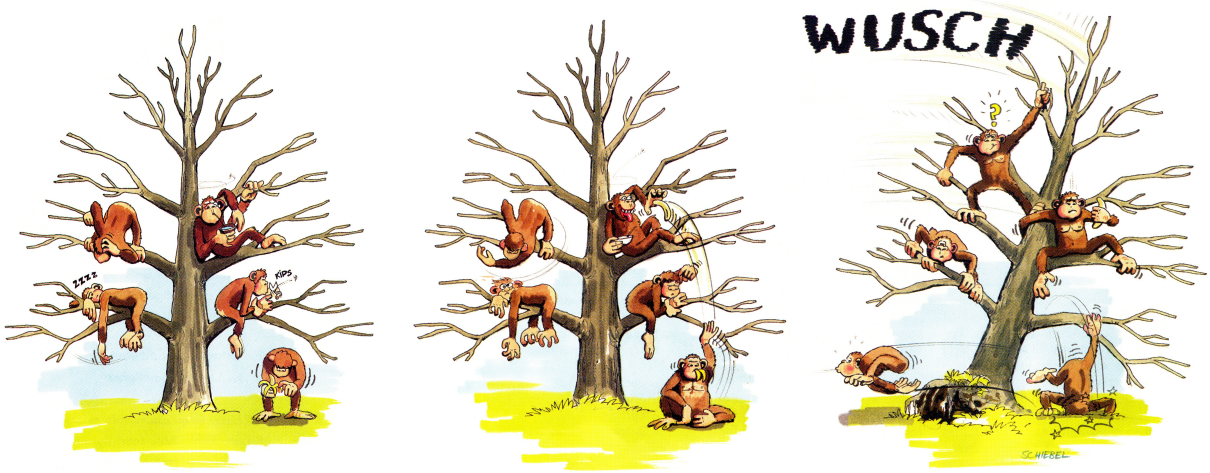
Ausbalancierte Bäume

Sortieren mit AVL-Bäumen

Das haut selbst den stärksten Baum um – der sprichwörtliche Griff nach der letzten Banane. Und wie die Balance durch eine Kleinigkeit gestört wird und zum Kippen des Baumes führt, so können bei Datenbanken zusätzlich einzufügende Schlüssel die Strukturen aus dem Lot bringen. Meist sind lange Suchzeiten die unerwünschte Folge. Sogenannte AVL-Bäume umgehen die Problematik.



Niemand stöbert gerne in ungeordneten Papierbergen. Wer sucht, der möchte die Gewünschte Information möglichst nach einem Griff zum richtigen Ordner in der Hand halten. Eine effiziente Organisation kann so eine Menge Zeit sparen. Gleiches gilt für Daten, die auf einer Festplatte abgespeichert sind: Je besser die Organisation, desto schneller kann auf die Daten zugegriffen werden. Sortieralgorithmen zählen daher zu dem Handwerkszeug, das eigentlich jeder gute Programmierer kennen sollte. Heapsort und Quicksort sind die Einsteigermodelle [1], doch es gibt auch solche für Fortgeschrittene. Diese Luxusversionen sind im allgemeinen an komplexere Datenstrukturen gebunden. Ein Beispiel sind die sogenannten AVL-Bäume, benannt nach ihren Schöpfern Adel'son-Vel'skii und Landis [2].

Finden und Löschen – ein Dilemma

Für sortierte Felder gibt es etliche schnelle Suchmethoden. Eine der schnellsten ist die Bisektion, zu deutsch Zweiteilung (*Bild 1*). Man beginnt mit der Suche am Anfang des Feldes. Ist der dort stehende Sortierschlüssel – also das Kriterium des Datensatzes, nach dem sortiert wird – kleiner als der gesuchte, teilt man das Feld in der Mitte. Ist der gesuchte Schlüssel kleiner als der mittige, springt man in die Mitte der linken Hälfte, sonst in die der rechten. Dieses Vorgehen -stetes Halbieren des Suchbereichs – wiederholt man, bis der gesuchte Schlüssel gefunden wurde oder das Feld nicht mehr geteilt werden kann; in diesem Fall existiert der gesuchte Schlüssel nicht.
Die Verwendung von Feldern, die ja einen direkten Zugriff auf bestimmte Feldelemente, etwa die Mitte, erlauben, kann aber schnell zu Problemen führen, insbesondere dann, wenn das Programm zu viele Variablen enthält und das Datensegment überläuft. Dazu kommt die bekannte Schwäche statischer Felder: Bei vielen Programmen kann man vorher nicht wissen, wieviel Speicher während der Laufzeit benötigt wird. Legt man sich auf eine Anzahl fest, etwa auf 1000 Datensätze, so kann es möglich sein, daß man alsbald 1001 Datensätze braucht. Andererseits wird unnötig viel Speicherplatz belegt, wenn sich herausstellt, daß nur wenige Datensätze anfallen. Ein anderes Problem tritt beim Löschen oder Einfügen von Daten auf. Beide Operationen benötigen relativ viel Zeit, da man alle nachfolgenden Feldelemente entsprechend verschieben muß.

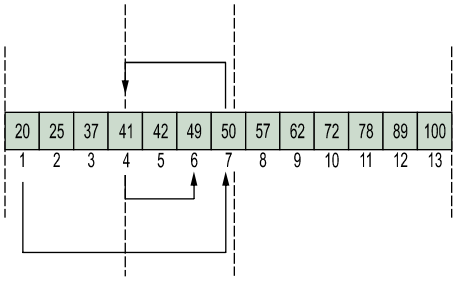


Bild 1. Bisektion oder Zweiteilung: schnelle Suche nach der 49

Schon ganz gut – Binärbäume

Um das angeschnittene Speicherproblem zu umgehen, greift man in der Praxis auf dynamische Zeigerkonstruktionen zurück, weicht auf den Heap aus und legt die Schlüssel in einer verketteten Liste ab. Nachteil: Es ist nicht mehr möglich, ein bestimmtes Element direkt anzusprechen, was für die meisten effizienten Sortier- und Suchverfahren aber Voraussetzung ist.

Beide Datenstrukturen – Feld und verkettete Liste – haben somit Vor- und Nachteile, eine optimale Struktur für Sortier- und Suchprobleme weisen sie jedoch bei weitem nicht auf.

Für solche Probleme eignet sich der binäre Suchbaum besser (*Bild 2*). Ein binärer Suchbaum, auch Binärbaum genannt, besteht aus sogenannten Knoten. Jeder Knoten enthält den Sortierschlüssel und einen Verweis auf den zugehörigen Datensatz. Natürlich könnte man in einem Knoten auch den gesamten Datensatz speichern, dies erfordert aber im allgemeinen unnötig viel Speicherplatz.

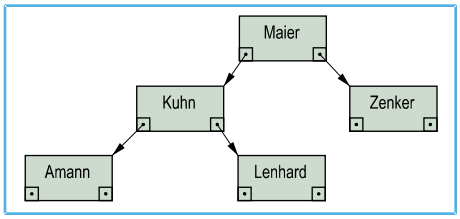


Bild 2. Binärer Suchbaum oder Binärbaum: die Datenstruktur zum Einfügen, Löschen und Suchen

Zwei Zeiger verweisen auf die Nachfolger des Knotens, die ebenfalls wieder Knoten sind (oder NIL = Ende). Diese Zeiger heißen auch Kanten oder Äste des Baumes. Man unterscheidet die rechten und linken Nachfolger eines Knotens. Der linke Nachfolger enthält einen Schlüssel, der kleiner ist als der seines Vorgängers, der rechte einen größeren. Der oberste Knoten des Baumes wird als Wurzel bezeichnet. Ein Knoten ohne Nachfolger heißt Blatt. Besteht der Baum nur aus der Wurzel, so ist der einzige Knoten des Baumes gleichzeitig Baum, Wurzel und Blatt. Der Nachfolger eines bestimmten Knotens zusammen mit seinen eventuell nicht vorhandenen Nachfolgern heißt auch Unterbaum des Knotens, spezieller: rechter oder linker Unterbaum.

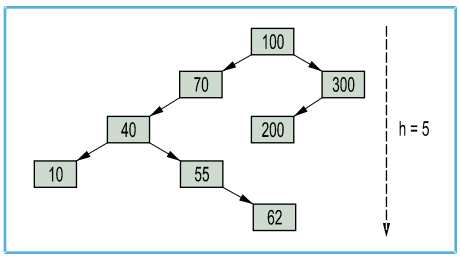


Bild 3. Höhe des Baumes: entscheidend für die Suchdauer

Die Höhe eines Unterbaumes ist die maximale Distanz zwischen der Wurzel und einem Blatt. Der leere, nichtexistente Baum hat somit die Höhe 0, ein Baum mit nur einem Knoten hat die Höhe 1, der Baum in *Bild 3* die Höhe 5.

Hangeln von Knoten zu Knoten

Das Suchschema in einem solchen Baum gestaltet sich recht einfach:

1. Falls der Schlüssel des aktuellen Knotens gleich dem gesuchten Schlüssel ist → Ende: Schlüssel gefunden.
2. Falls der aktuelle Knoten=NIL, falls man also aus dem Baum herausfällt → Ende: Der gesuchte Schlüssel existiert nicht.
3. Falls der Schlüssel im aktuellen Knoten größer als der gesuchte ist → Gehe nach links, ansonsten nach rechts.
4. Weiter bei Punkt 1.

Bevor man jedoch mit der fröhlichen Suche beginnen kann, muß man den Baum erst pflanzen: Wie geschieht also das Einfügen in einen Binärbaum? Ganz einfach: Genauso wie das Suchen, mit einem Unterschied: Fällt man aus dem Baum heraus, wird genau dort, wo man sich zuletzt im Baum befand, der neue Knoten mit dem neuen Schlüssel angehängt. War der neue Schlüssel kleiner als der letzte betrachtete, wird er links angehängt, sonst rechts. Der neue Knoten hat also keine Nachfolger und ist somit ein Blatt. Damit alles klappt, darf ein solcher Suchbaum keine zwei gleichen Schlüssel enthalten. Natürlich könnte man vereinbaren, daß nicht nur bei kleiner, sondern bei kleiner-gleich nach links verzweigt wird. Das führt bei binären Suchbäumen zu keinerlei Problemen, in AVL-Bäumen ist es jedoch nicht möglich, wie wir später sehen werden.

Im Gegensatz zum Einfügen oder Suchen gestaltet sich das Löschen im Binärbaum etwas schwieriger. Soll ein Blatt gelöscht werden, so ist das weiter kein Problem: Man schneidet es einfach ab. Auch das Entfernen eines Knotens mit nur einem Nachfolger ist nicht weiter schwierig: Der Vorgänger des betreffenden Knotens wird mit dessen Nachfolger verkettet und dann entfernt. Man sieht leicht, daß die Baumstruktur in diesem Fall immer erhalten bleibt (*Bild 4*).

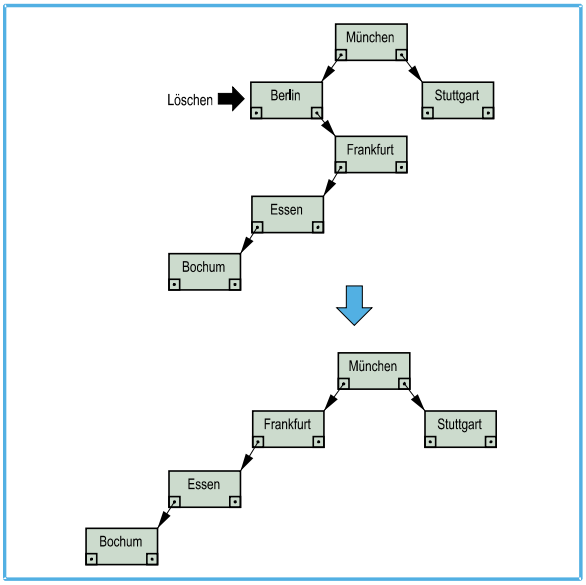


Bild 4. Problemlos: Löschen eines Knotens mit nur einem Nachfolger

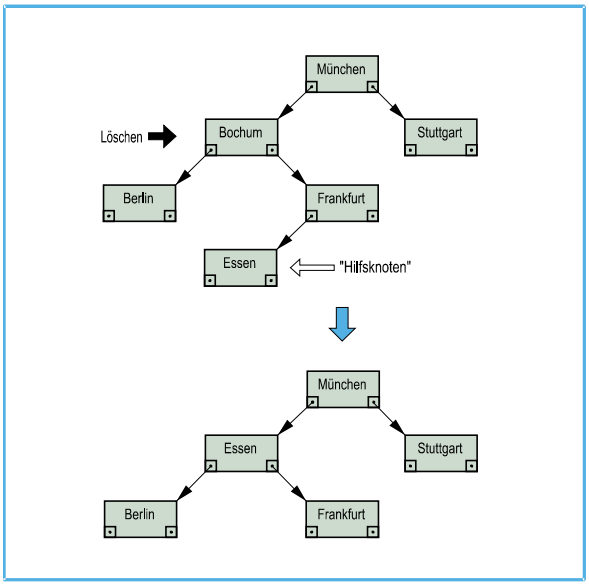


Bild 5. Trickreich: Löschen eines Knotens mit zwei Nachfolgern

Etwas übler sieht es aus, wenn ein Knoten gelöscht werden soll, der zwei Nachfolger hat. Man kann ihn nicht einfach rausnehmen, wie in den anderen beiden Fällen, da sein Vorgänger dann drei Nachfolger hätte. Man macht es daher anders und benutzt entweder den kleinsten Knoten im rechten Unterbaum des zu löschenden Knotens oder den größten im linken Unterbaum. Dieser Hilfsknoten hat höchstens einen Nachfolger: er hat entweder keinen linken Nachfolger (kleinster Knoten im rechten Unterbaum) oder keinen, rechten (größter Knoten im linken

Unterbaum). Er wird nun aus dem Baum herausgetrennt und an die Stelle des zu löschenden Knotens gesetzt, der entfernt wird. Die Baumstruktur bleibt durch die spezielle Wahl des Ersatzknotens erhalten (*Bild 5*).
Jetzt fehlt uns nur noch eine sortierte Ausgabe, so etwa: Der Baum wird mit einer rekursiven Prozedur durchkämmt:

```
procedure TraverseTree(Wurzel: Knoten);
begin
  if Wurzel=NIL then Exit;
  TraverseTree(Wurzel.Links);
  Ausgabe (Wurzel.DatenSatzNummer);
  TraverseTree(Wurzel.Rechts);
end;
```

Es gibt jedoch noch eine andere Möglichkeit. Dazu erhält jeder Knoten zusätzlich zu den Zeigern auf seine beiden Nachfolger einen Zeiger auf seinen Vorgänger. Vorteil: Es ist nun auch möglich, zu jedem beliebigen Knoten im Baum denjenigen zu finden, der bezüglich der Sortierreihenfolge den nächstgrößeren Schlüssel enthält. Das wird dann notwendig sein, wenn der Baum beispielsweise benutzergesteuert Knoten für Knoten durchkämmt werden muß. Und es ist auch dann vorteilhaft, wenn für jeden Knoten aufwendigere Prozeduren als lediglich die Ausgabe des zugehörigen Datensatzes aufgerufen werden müssen.
Der nächste Knoten ist folgendermaßen zu finden: Falls der aktuelle Knoten einen rechten Nachfolger hat, gehe zum kleinsten Knoten im rechten Unterbaum; sonst wandere solange nach oben, bis der aktuelle Knoten einen linken Nachfolger besitzt (vgl. dazu auch die Funktionen *NextNode* und *LastNode* im *Listing*). Dieses Vorgehen entspricht übrigens genau demjenigen der rekursiven Prozedur. Aufgrund der oben beschriebenen Vorteile wurde für die Unit AVLTREES.PAS auch die zweite Möglichkeit gewählt, der Baum mit Rückzeiger also.
Damit hätten wir alle Befehle, die wir für unsere Arbeit mit Bäumen brauchen:

- Suchen
- Einfügen
- Löschen
- Bearbeiten in Sortier-Reihenfolge.

Bis auf das Löschen sind diese Operationen einfach und schnell. *Bei idealer Gestalt des Baumes* benötigt man zum Finden eines bestimmten Knotens in einem Baum mit n Knoten $\log_2(n)+1$ Vergleiche. Das heißt, bei einem Baum mit 2048 Knoten braucht der Rechner im schlechtesten Fall 12 Vergleiche (den letzten, der die Gleichheit feststellt, mit eingeschlossen), um einen bestimmten Knoten zu finden.
Nicht umsonst ist *bei idealer Gestalt des Baumes* kursiv gesetzt. Diese ideale Gestalt tritt beim binären Suchbaum nur äußerst selten auf, und wenn, dann zufällig. Viel wahrscheinlicher ist es, daß der Baum recht rasch eine aufs übelste degenerierte Gestalt annimmt. Wenn zum Beispiel ein skrupelloser Benutzer – und skrupellos sind sie ALLE – hingeht und die zu sortierenden Daten bereits in einer beinahe vollständig sortierten Reihenfolge eingibt, nähert sich das Aussehen des Baumes dem einer Liste (*Bild 6*). Die ganze Mühe, die wir uns mit unserem Baum gemacht haben, ist futsch, denn das Suchen dauert jetzt fast so lange wie die sequentielle Suche in einem Feld oder einer verketteten Liste.

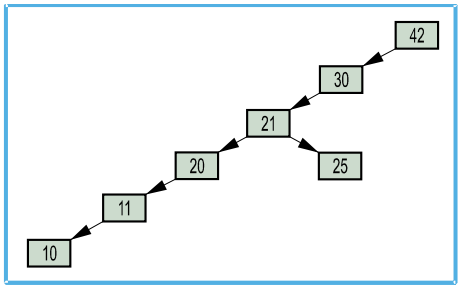


Bild 6. Degeneriert: listenartiger „Strauch“

Natürlich kann man das verhindern, indem man etwa durch Mischen der Daten vor dem Eingeben dafür sorgt, daß die Eingabe-Reihenfolge mehr oder weniger zufällig wird. Aber wer zerhackstückt schon gerne seine sorgfältig eingegebenen Daten – hier liegt eine gewisse psychologische Hemmschwelle; irgendwie scheint das nicht „sauber" und schließlich soll der Computer ja auch nicht rumzicken.

Der rettende Einfall

Zu diesem Problem haben sich Adel'son-Velskii und Landis bereits 1962 Gedanken gemacht. Ihre Idee war folgende: Um den Baum immer möglichst nahe seiner ausgeglichenen, balancierten Idealform (beide Unterbäume eines jeden Knotens haben annähernd dieselbe Höhe) zu halten, wird beim Einfügen und Löschen etwas mehr Zeit investiert, was sich später bei allen Operationen auszahlt. Dafür ist es notwendig, in jedem Knoten des Baumes noch zusätzlich den sogenannten Balance-Faktor zu speichern. Dieser Balance-Faktor ist gerade die Differenz aus der Höhe des linken Unterbaumes und der Höhe des rechten Unterbaumes.
In einem AVL-Baum sind per Definition nur Knoten enthalten, die den Balance-Faktor -1, 0 oder +1 haben – andernfalls ist der Baum kein AVL-Baum. Mit anderen Worten: In einem AVL-Baum ist für jeden Knoten die Höhendifferenz zwischen linkem und rechtem Unterbaum maximal 1. So ist zum Beispiel der Baum in *Bild 7a* ein AVL-Baum, der in *Bild 7b* jedoch nicht.

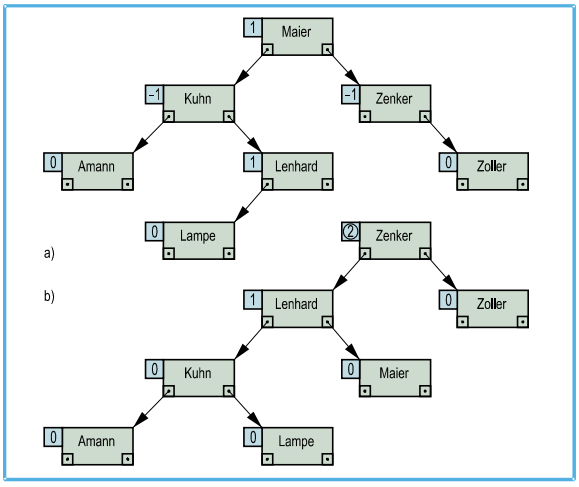


Bild 7. Ausbalanciert oder nicht?
a) Beispiel für einen AVL-Baum
b) Gegenbeispiel

Sobald in einem Knoten der Balance-Faktor den Wert +1 überschreitet oder den Wert -1 unterschreitet muß eingegriffen werden: Der entsprechende Unterbaum wird *rotiert* Was das bedeutet, dazu später mehr.
Zunächst untersuchen wir die Vorgänge beim Einfügen in einen AVL-Baum anhand eines Beispiels. In *Bild 8* sehen wir einen ganz einfachen AVL-Baum mit Höhe 1. Fügen wir nun noch einen Knoten mit dem Schlüssel *Andreas* ein. Da ein neu eingefügter Knoten immer ein Blatt ist, erhält dieser den Balance-Faktor Null (Höhe des linken U-Baums = Höhe des rechten U-Baums = Null). Nach dem Einfügen gehen wir in Richtung Wurzel (indem wir die vorher – welch ein günstiger Zufall – eingeführten Rückzeiger benutzen) und aktualisieren entlang dieses Weges die Balance-Faktoren der einzelnen Knoten, und zwar nach folgenden Schema: Sind wir von links in einen Knoten gekommen, wird der Balance-Faktor dieses Knotens um eins erhöht, denn durch das Einfügen hat sich die Höhe seines linken Unterbaumes um eins erhöht. Dementsprechend erniedrigen wir den Balance-Faktor eines Knotens, in den wir von rechts kommen, um eins.

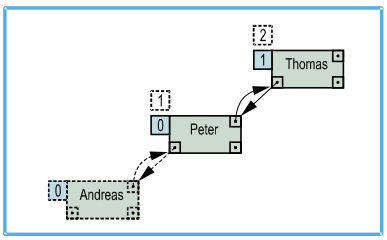


Bild 8. Gleichgewichtsstörung: Wer einfügt muß auch ausbalancieren

Wenn der Balance-Faktor eines Knotens auf unserem Weg Null wird, sind wir fertig, denn dann hat sich die Höhe des Unterbaums, dessen Wurzel der betreffende Knoten darstellt, nicht verändert. Erreichen wir die Wurzel, sind wir ebenfalls fertig und brauchen sonst nichts zu tun.

Bäumchen dreh' dich

Wird ein Balance-Faktor auf unserem Weg jedoch +2 oder -2, müssen wir etwas tun. In unserem Beispiel ist dies der Fall, wenn wir die Wurzel erreichen. Ein Knoten, dessen Balance-Faktor +2 oder -2 ist, heißt kritischer Knoten. Um einen solchen kritischen Knoten muß nun rotiert werden. Wie das geschieht, ersieht man am besten aus *Bild 9*. Eine solche Rotation nennt man einfache Rechtsrotation. In dem Beispiel in *Bild 10* muß anders rotiert werden; es handelt sich dabei um eine doppelte Rechtsrotation. Es gibt eine allgemeine Regel, in welchem Fall wie rotiert werden muß: Ist der Balance-Faktor des kritischen Knotens +2, der Unterbaum also linkslastig, muß nach rechts rotiert werden. Ist der Balance-Faktor des linken Unterbaums des kritischen Knotens +1, genügt eine einfache Rotation, ist er -1, muß doppelt rotiert werden. Entsprechendes gilt seitenverkehrt für einen kritischen Knoten mit Balance-Faktor -2. Auch hier wird bei gleichem Vorzeichen der beiden Knoten einfach, bei verschiedenen Vorzeichen doppelt rotiert. Nach dem Rotieren ist der Balance-Faktor der Wurzel des Unterbaumes immer Null, so daß wir an dieser Stelle mit dem Einfügungsvorgang endgültig fertig sind.

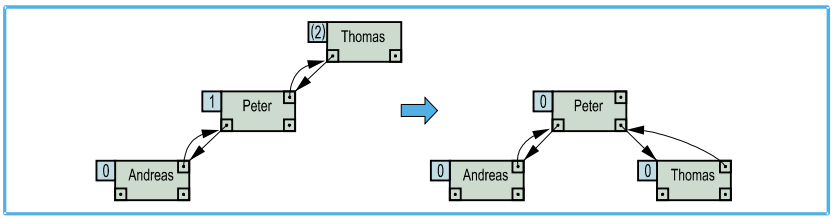


Bild 9. Rotation: Garantie für die AVL-Struktur des Baumes

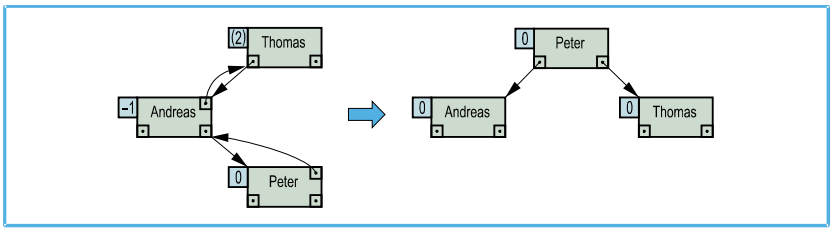


Bild 10. Doppelte Rotation: das Vorzeichen des Balancefaktors sowie die Unterbaumstruktur legen die Rotationsrichtung fest

Wie das Rotieren im einzelnen geschieht, ersieht man am besten aus dem Listing. Aus den Bildchen wird jedenfalls schon klar, daß ein AVL-Baum keine zwei gleichen Schlüssel enthalten darf. Wenn man dieselbe Kleiner-gleich-Vereinbarung wie bei binären Suchbäumen benutzt, fällt man damit früher oder später auf die Nase. Ein Knoten mit dem gleichen Schlüssel wie sein Vorgänger kann vor dem Rotieren dessen rechter und danach sein linker Nachfolger.

Beim Löschen wird die ganze Sache noch wesentlich komplizierter. Es kann notwendig sein, mehrmals zu rotieren, weil bestimmte Sonderfälle auftreten können – zum Beispiel ein kritischer Knoten mit Balance-Faktor -2, dessen linker Nachfolger den Balance-Faktor 0 hat. Außerdem ist es beim Löschen nicht unbedingt so, daß der Balance-Faktor eines kritischen Unterbaums nach dem Rotieren Null ist.

Diese verzwickten Einzelheiten möchte ich dem Leser aber ersparen und statt dessen zum Schluß dieses Beitrages noch etwas auf die vorgestellte Unit – um die geht's nämlich, ob man's glaubt oder nicht – eingehen. Nur noch ein Wort zum AVL-Baum. Mit AVL-Bäumen kommt man zu jeder Zeit, unabhängig von der Anzahl der Knoten, der Idealform des Baumes so nahe wie möglich. Nur so nutzt man die Vorteile des Suchbaumes wirklich aus. Das Finden eines Knotens in einem AVL-Baum benötigt bei n Knoten im schlechtesten Fall, wenn also der gesuchte Knoten ganz unten im Baum steht, $\log_2(n) + 2$ Vergleiche.

Nach dem Vorspiel geht's zur Sache

Die Unit AVLTREES.PAS wurde unter TURBO-PASCAL 4.0 erstellt, kann aber ohne weiteres unter TURBO-PASCAL 5.0 oder 5.5 verwendet werden. Das Umschreiben in eine Include-Datei für TURBO-PASCAL 3.0 ist ohne großen Aufwand möglich, am eigentlichen Code muß nichts geändert werden, lediglich die uses-Anweisungen, die Compiler-Direktiven und die Worte interface und implementation fallen weg.

Ein Programm, das die Unit verwendet, braucht eine feste Variable des Typs AVL-Tree. Diese Variable, die beispielsweise AVLRoot heißen könnte, ist ein Zeiger auf die Wurzel des Baumes und muß mit AVL-Root:=NIL

initialisiert werden, und zwar noch vor jedem anderen Aufruf einer Prozedur/Funktion der Unit.
Die Unit sollte für jedes beliebige Programm verwendbar sein. Daraus ergab sich die Struktur des Knotens (type AvlNode). Außer den Standard-Elementen, also Balance-Faktor, Rückzeiger und Zeiger auf linken oder rechten Nachfolger, enthält ein Knoten ein Array des Typs Byte und einen Zeiger auf ein weiteres Array des Typs Byte. Im ersten, kleineren Array wird der Schlüssel abgelegt. Für ihn kann jeder einfache Datentyp verwendet werden, außer Boolean, was nicht sehr sinnvoll wäre, da ein solcher Schlüssel nur zwei Werte annehmen kann. Auch Strings können als Schlüssel verwendet werden. Welcher Typ für den Schlüssel verwendet wird, muß in der Variablen KeyType festgelegt werden, und zwar vor dem ersten Aufruf einer Prozedur der Unit. Die Variable KeyType kann folgende Werte annehmen: KChar, KByte, KInteger, KReal und KString. Eine nähere Erklärung hinsichtlich der Bedeutung dieser Werte erübrigt sich. Außerdem benötigt die Unit noch zwei weitere Informationen, nämlich die Größe des Schlüssels, die in der Variablen KeySize abgelegt werden muß, und die Größe der Daten. Die Daten sind die zum Schlüssel gehörenden Daten, normalerweise ein Verweis, etwa eine Datensatznummer. Es können aber auch die vollständigen Daten sein, beispielsweise eine ganze Adresse.
Der Datentyp kann frei gewählt werden. Die Größe der Daten (in Bytes, kann man mit SizeOf(Typ) bestimmen) muß der Unit jedoch mitgeteilt werden. Sie wird in der Variablen DatSize abgelegt. Die beiden Variablen KeySize und DatSize müssen vor dem ersten Aufruf einer Prozedur der Unit initialisiert werden. Falls die Daten länger sind als 1024 Bytes, muß die Konstante DatMax entsprechend angepaßt werden.
Der Wert der Konstanten DatMax oder Key-Max hat keinen Einfluß auf den Speicherplatz, den ein Knoten auf dem Heap beansprucht. Es wird entsprechend der Schlüsselbeziehungsweise Datengröße nur soviel Speicher belegt, wie tatsächlich erforderlich ist, weshalb die Befehle GetMem und Free-Mem anstelle von New und Dispose verwendet werden.
Es ist möglich, mit der Unit mehrere Bäume gleichzeitig zu bearbeiten. Man benötigt dann für jeden Baum eine eigene Variable vom Typ AvlTree. Wenn die Schlüssel- oder Datengröße dieser verschiedenen Bäume unterschiedlich ist, müssen vor jedem Baumwechsel die Variablen KeySize, DatSize und KeyType entsprechend gesetzt werden, sonst ist ein Absturz unvermeidlich.

Finger weg vom Baum!

Die einzelnen Prozeduren und Funktionen der Unit sind in den Kommentaren erklärt. Jede Prozedur/Funktion enthält einen Parameter des Typs AvlTree. In diesem Parameter wird außer bei den Funktionen NextNode und LastNode die Wurzel des AVL-Baums übergeben. Ein AVL-Baum ist ein sehr empfindliches Gebilde. Deshalb empfiehlt es sich nicht, an diese Prozeduren andere Parameter zu übergeben, wie zum Beispiel irgendwelche Knoten mitten im Baum. Damit man sicher sein kann, daß nichts schief geht, sollte man auf die Bäume ausschließlich mit den Funktionen der Unit zugreifen. Es gibt aber trotzdem einige Prozeduren, die man gefahrlos umschreiben oder hinzufügen kann. Diese Prozeduren dürfen jedoch folgende Dinge nicht tun:

- Den Aufbau des Baumes ändern, also irgendwelche Zeiger innerhalb des Baumes verbiegen.
- Schlüssel im Baum verändern.
- Irgendwelche Knoten hinzufügen oder entfernen.
- Balance-Faktoren verändern.

Das Programm AVLDEMO.PAS enthält alle Aufrufe der Unit mindestens einmal. Es zeigt das Zusammenspiel von Schlüsseln und Verweisen anhand einer sehr einfachen Adressverwaltung, die auf viele professionelle Abfangroutinen verzichtet. Wer sich hier austoben möchte, der kommt sicher auf seine Kosten.

Philipp Leibfried/ks

Literatur

[1] *Press, W. H., Flannery, B. P., Teulkowski, S. A., Vetterling, W. T.*: Numerical Recipes in Pascal. Cambridge University Press, 1989.
[2] *Adel'son-Velskü, G. M., Landis, Y. M.*: Soviet Math. Dokl, S. 1259-1262 (1962).
[3] *Manber, U.*: Introduction to Algorithms. Addison-Wesley, 1989.

Listing 1: AVL-Unit

```
unit AVLTrees;
{*******************************************}
{ Written in August '90 by P. Leibfried     }
{ (C) 1990 by Philipp Leibfried             }
{*******************************************}
{$R+,S+,I+,D+,F-,V-,B-,N-,L+}
{$M 16384,0,655360}
{*******************************************}
interface
{*******************************************}
uses Crt;
const
  KeyMax = 256;
  DatMax = 1024;
type
  AvlTree  = ^AvlNode;
  Balance  = (Min2, Min1, Zero, Pl1, Pl2);
  KType    = (KByte, KChar, KInteger, KReal, KString);
  KeyArray = array [1..KeyMax] of byte;
  DatPtr   = ^DatArray;
  DatArray = array [1..DatMax] of byte;
  AvlNode  = record
               BalFact         : Balance;
               Left,Right,Last : AvlTree;
               Data            : DatPtr;
               TreeKey         : KeyArray;
             end;
var
  KeySize : Integer;
{ Statusvariablen KeySize und KeyType dürfen   }
{ nur einmal gesetzt werden, solange der Baum  }
{ nicht wieder gelöscht wird                   }
  KeyType : KType;
  DatSize : Integer;
{*═══════════════════════════════════════════*}
function Height(HTree: AVLTree) : Integer;
{ Berechnet Höhe des Baumes HTree              }
{*═══════════════════════════════════════════*}
function Lowest(Var LData;
                    BegTree: AvlTree): AvlTree;
{ Gibt Zeiger auf Knoten mit niedrigstem       }
{ Schlüssel im (Unter-)Baum BegTree zurück.    }
{ Ist BegTree=NIL, so wird NIL zurückgegeben.  }
{ Knotendaten stehen in LData.                 }
{*═══════════════════════════════════════════*}
function Highest(Var HData;
                     BegTree: AvlTree): AvlTree;
{ Gibt Zeiger auf den Knoten mit höchstem      }
{ Schlüssel im (Unter-)Baum BegTree zurück.    }
{ Ist BegTree=NIL, so wird NIL zurückgegeben.  }
{ Knotendaten stehen in HData.                 }
{*═══════════════════════════════════════════*}
function NextNode(Var NData;
                      BegTree: AvlTree): AvlTree;
{ Gibt Zeiger auf den auf nächsthöheren        }
{ BegTree-Schlüssel zurück. Falls nicht        }
```

```
{ existent wird, NIL zurückgegeben.            }
{ Knotendaten stehen in NData                  }
{*═══════════════════════════════════════════*}
function LastNode(Var LData;
                      BegTree : AvlTree): AvlTree;
{ Gibt Zeiger auf den auf nächstniedrigeren    }
{ BegTree-Schlüssel zurück. Falls nicht        }
{ existent, wird NIL zurückgegeben.            }
{ Knotendaten stehen in LData                  }
{*═══════════════════════════════════════════*}
function SearchNode(Var SKey, SData;
                        BegTree: AvlTree): AvlTree;
{ Gibt Zeiger auf Knoten mit Schlüssel SKey    }
{ zurück. Falls nicht existent, wird NIL       }
{ zurückgegeben. Knotendaten stehen in SData;  }
{ In BegTree wird normalerweise die Wurzel     }
{ des Baumes übergeben. Es ist jedoch auch     }
{ möglich, von einem bestimmten Knoten im      }
{ Baum aus suchen zu lassen.                   }
{*═══════════════════════════════════════════*}
procedure InsertNew(Var NewKey, NewData;
                    Var InsTree: AvlTree;
                    Var Success: boolean);
{ Enthält neuen Schlüssel, neue Daten und die  }
{ Wurzel von InsTree. In diesem Baum wird ein  }
{ neuer Knoten mit NewKey und NewData angelegt }
{ Falls der Schlüssel im Baum schon vorhanden  }
{ ist, wird kein Knoten eingefügt. Success     }
{ wird dann - und nur dann - auf FALSE gesetzt.}
{*═══════════════════════════════════════════*}
procedure ReplaceData(Var OldKey, NewData;
                          RepTree: AvlTree;
                       Var Success: boolean);
{ Sucht in RepTree Knoten mit Schlüssel OldKey.}
{ Falls nicht existent, geschieht weiter       }
{ nichts, Success wird auf FALSE gesetzt.      }
{ Falls existent werden die Knotendaten        }
{ durch NewData ersetzt.                       }
{*═══════════════════════════════════════════*}
procedure DeleteNode(Var DelKey;
                     Var DelTree: AvlTree;
                     Var Success: boolean);
{ Löscht Knoten mit Schlüssel DelKey im Baum   }
{ mit Wurzel DelTree. Falls Schlüssel nicht    }
{ existent, kann auch nichts gelöscht werden.  }
{ Success wird nur dann auf FALSE gesetzt.     }
{*═══════════════════════════════════════════*}
procedure DelAllTree(Var DelTree : AvlTree);
{ Löscht den ganzen Baum mit Wurzel DelTree.   }
{ DelTree wird NIL.                            }
{*******************************************}
implementation
{*******************************************}
type  TreeDirec =  (FromL,FromR);
      CompResult = (Equal,Bigger,Small);
{*═══════════════════════════════════════════*}
function Compare(Var Key1,Key2) : CompResult;
  { Vergleichsfunktion für alle einfachen      }
```

```
  { Datentypen und Strings.                    }
var   By1 : Byte     absolute Key1;
      By2 : Byte     absolute Key2;
      In1 : Integer  absolute Key1;
      In2 : Integer  absolute Key2;
      Rl1 : Real     absolute Key1;
      Rl2 : Real     absolute Key2;
      St1 : String   absolute Key1;
      St2 : String   absolute Key2;
begin
  case KeyType of
    KChar, KByte : if By1<By2 then
                     Compare:=Small
                   else if By1>By2 then
                     Compare:=Bigger
                   else  Compare:=Equal;
    KInteger     : if In1<In2 then
                     Compare:=Small
                   else if In1>In2 then
                     Compare:=Bigger
                   else  Compare:=Equal;
    KReal        : if Rl1<Rl2  then
                     Compare:=Small
                   else if Rl1>Rl2  then
                     Compare:=Bigger
                   else  Compare:=Equal;
    KString      : if St1<St2  then
                     Compare:=Small
                   else if St1>St2 then
                     Compare:=Bigger
                   else  Compare:=Equal;
  end;
end;
{*═══════════════════════════════════════════════*}
function Height(HTree: AvlTree): Integer;
var  H1,H2 : Integer;
begin
  if HTree=NIL then
  begin
    Height:=0;
    Exit;
  end;
  H1:=Height(HTree^.Left);
  H2:=Height(HTree^.Right);
  if H1>H2 then Height:=Succ(H1)
  else Height:=Succ(H2);
end;
{*═══════════════════════════════════════════════*}
function Lowest(Var LData;
                  BegTree: AvlTree): AvlTree;
begin
  if BegTree=nil then
  begin
    Lowest:=nil;
    Exit;
  end;
  while BegTree^.Left<>nil do
    BegTree:=BegTree^.Left;
```

```
  Move(BegTree^.Data^, LData, DatSize);
  Lowest:=BegTree;
end;
{*═══════════════════════════════════════════════*}
function Highest(Var HData;
                     BegTree : AvlTree) : AvlTree;
begin
  if BegTree=nil then
  begin
    Highest:=nil;
    Exit;
  end;
  while BegTree^.Right<>nil do
    BegTree:=BegTree^.Right;
  Move(BegTree^.Data^, HData, DatSize);
  Highest:=BegTree;
end;
{*═══════════════════════════════════════════════*}
Function NextNode(Var NData;
                      BegTree: AvlTree): AvlTree;
var   UpperT : AvlTree;
begin
  if BegTree=nil then
  begin
    NextNode:=nil;
    Exit;
  end;
  if BegTree^.Right<>nil then
    NextNode:=Lowest(NData,BegTree^.Right)
  else
  begin
    UpperT:=BegTree;
    repeat
      BegTree:=UpperT;
      UpperT:=UpperT^.Last;
    until (UpperT=nil) or (UpperT^.Left=BegTree);
    if UpperT<>nil then
      Move(UpperT^.Data^, NData, DatSize)
    else
      FillChar(NData,DatSize,0);
    NextNode:=UpperT;
  end;
end;
{*═══════════════════════════════════════════════*}
function LastNode(Var LData;
                      BegTree: AvlTree): AvlTree;
var   UpperT : AvlTree;
begin
  if BegTree=nil then
  begin
    LastNode:=nil;
    Exit;
  end;
  if BegTree^.Left<>nil then
    LastNode:=Highest(LData,BegTree^.Left)
  else
  begin
    UpperT:=BegTree;
```

```
    repeat
      BegTree:=UpperT;
      UpperT:=UpperT^.Last;
    until (UpperT=nil) or (UpperT^.Right=BegTree);
    if UpperT<>nil then
      Move(UpperT^.Data^, LData, DatSize)
    else
      FillChar(LData,DatSize,0);
    LastNode:=UpperT;
  end;
end;
{*═══════════════════════════════════════════════*}
function SearchNode(Var SKey, SData;
                        BegTree: AvlTree): AvlTree;
begin
  if BegTree=nil then
  begin
    SearchNode:=nil; Exit;
  end;
  repeat
    case Compare(SKey,BegTree^.TreeKey) of
      Equal  : begin
                 Move(BegTree^.Data^, SData, DatSize);
                 SearchNode:=BegTree;
                 Exit;
               end;
      Small  : BegTree:=BegTree^.Left;
      Bigger : BegTree:=BegTree^.Right;
    end;
  until BegTree=nil;
  FillChar(SData,DatSize,0);
  SearchNode:=NIL;
end;
{*═══════════════════════════════════════════════*}
procedure ReplaceData(Var OldKey, NewData;
                          RepTree: AvlTree;
                      Var Success: boolean);
var   RepNode : AvlTree;
      Dummy   : DatArray;
begin
  RepNode:=SearchNode(OldKey, Dummy, RepTree);
  if RepNode=NIL then
  begin
    Success:=false;
    Exit;
  end;
  Success:=true;
  Move(NewData, RepNode^.Data^, DatSize);
end;
{*********************************************}
{* Prozeduren zur Erhaltung der AVL-Struktur  *}
{*********************************************}
procedure AllBalanceFactors(AVL: AvlTree;
                            Var Height: Integer);
var   B, HLeft, HRight : Integer;
begin
  if AVL=nil then
  begin
```

```
    Height:=0;
    Exit;
  end;
  AllBalanceFactors(AVL^.Left, HLeft);
  AllBalanceFactors(AVL^.Right, HRight);
  B:=HLeft-HRight;
  AVL^.BalFact:=Balance(Byte(B+2));
  if B<0 then
    Height:=Succ(HRight)
  else
    Height:=Succ(HLeft);
end;
{*═══════════════════════════════════════════════*}
procedure SingleRotLeft(Var CriticalAVL: AvlTree);
var   Crit,CritR,CritRL : AvlTree;
begin
  Crit:=CriticalAVL;
  CritR:=Crit^.Right;
  CritRL:=CritR^.Left;
  CritR^.Last:=Crit^.Last;
  if Crit^.Last<>nil then
  begin
    if Crit^.Last^.Left=Crit then
      Crit^.Last^.Left:=CritR
    else
      Crit^.Last^.Right:=CritR;
  end;
  CriticalAVL:=CritR;
  CritR^.Left:=Crit;
  Crit^.Last:=CritR;
  Crit^.Right:=CritRL;
  if CritRL<>nil then
    CritRL^.Last:=Crit;
end;
{*═══════════════════════════════════════════════*}
procedure SingleRotRight(Var CriticalAVL: AvlTree);
var   Crit, CritL, CritLR: AvlTree;
begin
  Crit:=CriticalAVL;
  CritL:=Crit^.Left;
  CritLR:=CritL^.Right;
  CritL^.Last:=Crit^.Last;
  if Crit^.Last<>nil then
  begin
    if Crit^.Last^.Left=Crit then
      Crit^.Last^.Left:=CritL
    else
      Crit^.last^.Right:=CritL;
  end;
  CriticalAVL:=CritL;
  CritL^.Right:=Crit;
  Crit^.Last:=CritL;
  Crit^.Left:=CritLR;
  if CritLR<>nil then
    CritLR^.Last:=Crit;
end;
{*═══════════════════════════════════════════════*}
procedure DoubleRotRight(Var CriticalAVL: AvlTree);
```

```
var   Crit, CritL, CritLR : AvlTree;
      CLRR, CLRL          : AvlTree;
begin
  Crit:=CriticalAVL;
  CritL:=Crit^.Left;
  CritLR:=CritL^.Right;
  CLRL:=CritLR^.Left;
  CLRR:=CritLR^.Right;
  CritLR^.Last:=Crit^.Last;
  if Crit^.Last<>nil then
  begin
    if Crit^.Last^.Left=Crit then
      Crit^.Last^.Left:=CritLR
    else
      Crit^.Last^.Right:=CritLR;
  end;
  CriticalAVL:=CritLR;
  CritLR^.Right:=Crit;
  Crit^.Last:=CritLR;
  CritLR^.Left:=CritL;
  CritL^.Last:=CritLR;
  CritL^.Right:=CLRL;
  Crit^.Left:=CLRR;
  if CLRL<>nil then
    CLRL^.Last:=CritL;
  if CLRR<>nil then
    CLRR^.Last:=Crit;
end;
{*═══════════════════════════════════════════════*}
procedure DoubleRotLeft(Var CriticalAVL: AvlTree);
var   Crit, CritR, CritRL : AvlTree;
      CRLL,CRLR           : AvlTree;
begin
  Crit:=CriticalAVL;
  CritR:=Crit^.Right;
  CritRL:=CritR^.Left;
  CRLL:=CritRL^.Left;
  CRLR:=CritRL^.Right;
  CritRL^.Last:=Crit^.Last;
  if Crit^.Last<>nil then
  begin
    if Crit^.Last^.Left=Crit then
      Crit^.Last^.Left:=CritRL
    else
      Crit^.Last^.Right:=CritRL;
  end;
  CriticalAVL:=CritRL;
  CritRL^.Left:=Crit;
  Crit^.Last:=CritRL;
  CritRL^.Right:=CritR;
  CritR^.Last:=CritRL;
  Crit^.Right:=CRLL;
  CritR^.Left:=CRLR;
  if CRLL<>nil then
    CRLL^.Last:=Crit;
  if CRLR<>nil then
    CRLR^.Last:=CritR;
end;
```

```
{*═══════════════════════════════════════════════*}
procedure BalanceTree(Var CrTree: AvlTree);
var   Dummy : Integer;
{ Entscheidet, wie der (Unter-)Baum mit        }
{ Wurzel CrTree rotiert werden muß. Dies       }
{ kann anhand der Balance-Faktoren             }
{ eindeutig entschieden werden.                }
begin
  if (CrTree^.BalFact>Zero) and (CrTree^.Left^.BalFact<Zero) then
    DoubleRotRight(CrTree)
  else
    if (CrTree^.BalFact<Zero) and (CrTree^.Right^.BalFact>Zero) then
      DoubleRotLeft(CrTree)
    else
      if (CrTree^.BalFact>Zero) and (CrTree^.Left^.BalFact>=Zero) then
        SingleRotRight(CrTree)
      else
        if (CrTree^.BalFact<Zero) and (CrTree^.Right^.BalFact<=Zero)
then
          SingleRotLeft(CrTree);
  AllBalanceFactors(CrTree, Dummy);
end;
{*═══════════════════════════════════════════════*}
function MakeNewLeave(Var NewKey, NewData): AvlTree;
var NewNode : AvlTree;
{ Erzeugt einen neuen Knoten                   }
begin
  GetMem(NewNode,17+KeySize);
  with NewNode^ do
  begin
    BalFact:=Zero;
    GetMem(Data,DatSize);
    Left:=nil; Last:=nil; Right:=nil;
    Move(NewKey, TreeKey, KeySize);
    Move(NewData, Data^, DatSize);
  end;
  MakeNewLeave:=NewNode;
 end;
{*═══════════════════════════════════════════════*}
procedure InsertNew(Var NewKey,NewData;
                    Var InsTree: AvlTree;
                    Var Success: boolean);
var   SeekT, HelpT: AvlTree;
      Comp        : CompResult;
begin
  SeekT:=InsTree;
  HelpT:=nil;
  Success:=true;
  while SeekT<>nil do
  begin
    HelpT:=SeekT;
    Comp:=Compare(NewKey, SeekT^.TreeKey);
    case Comp of
      Equal  : begin
                 Success:=false;
                 Exit;
               end;
      Small  : SeekT:=SeekT^.Left;
```

```
      Bigger : SeekT:=SeekT^.Right;
    end;
  end;
  SeekT:=MakeNewLeave(NewKey, NewData);
  if HelpT=nil then
  begin
    InsTree:=SeekT;
    Exit;
  end;
  SeekT^.Last:=HelpT;
  case Comp of
    Small, Equal : HelpT^.Left:=SeekT;
    Bigger       : HelpT^.Right:=SeekT;
  end;
  repeat
    HelpT:=SeekT;
    SeekT:=SeekT^.Last;
    with SeekT^ do
    begin
      if HelpT=Left then
        BalFact:=Succ(BalFact)
      else BalFact:=Pred(BalFact);
      if (BalFact=Min2) or (BalFact=Pl2) then
      begin
        if SeekT=InsTree then
        begin
          BalanceTree(InsTree);
          SeekT:=InsTree;
        end
        else
          BalanceTree(SeekT);
      end;
    end;
  until (SeekT^.BalFact=Zero) or (SeekT^.Last=nil);
end;
{*═══════════════════════════════════════════════*}
procedure RemoveNode(Var DelTree, BTree: AvlTree;
                         DTree: AvlTree);
var SucTree,HelpT : AvlTree;
    Transfer      : DatArray;
begin
  if DTree=nil then
    Exit;
  if (DTree^.Left=nil) and (DTree^.Right=nil) then
  begin
    if DTree^.Last=nil then
    begin
      DelTree:=nil;
      BTree:=nil;
    end
    else
      with DTree^.Last^ do
        if Left=DTree then Left:=nil
        else Right:=nil;
      BTree:=DTree^.Last;
      FreeMem(DTree^.Data,DatSize);
      FreeMem(DTree,17+KeySize);
      DTree:=nil;
```

```
  end
  else
  if (DTree^.Left<>nil) xor (DTree^.Right<>nil) then
  begin
    if DTree^.Left=nil then
      SucTree:=DTree^.Right
    else
      SucTree:=DTree^.Left;
    if DTree^.Last<>nil then
      with DTree^.Last^ do
        if Left=DTree then Left:=SucTree
        else Right:=SucTree;
    SucTree^.Last:=DTree^.Last;
    if DTree=DelTree then
      DelTree:=SucTree;
    FreeMem(DTree^.Data,DatSize);
    FreeMem(DTree,17+KeySize);
    DTree:=nil;
    BTree:=SucTree;
  end
  else
    if (DTree^.Left<>nil) and (DTree^.Left<>nil) then
    begin
      { HelpT = Hilfsknoten                    }
      HelpT:=Lowest(Transfer, DTree^.Right);
      Move(HelpT^.TreeKey, DTree^.TreeKey, KeySize);
      Move(Transfer,DTree^.Data^, DatSize);
      RemoveNode(DelTree, BTree, HelpT);
    end;
end;
{*═══════════════════════════════════════════════*}
procedure DeleteNode(Var DelKey;
                     Var DelTree: AvlTree;
                     Var Success: boolean);
var BalTree,DTree : AvlTree;
    Comp          : CompResult;
    Dummy         : Integer;
begin
  DTree:=DelTree;
  Success:=true;
  Comp:=Compare(DelKey, DTree^.TreeKey);
  while (DTree<>nil) and (Comp<>Equal) do
  begin
    case Comp of
      Small  : DTree:=DTree^.Left;
      Bigger : DTree:=DTree^.Right;
    end;
    Comp:=Compare(DelKey,DTree^.TreeKey);
  end;
  if DTree=nil then
  begin
    Success:=false;
    Exit;
  end;
  RemoveNode(DelTree, BalTree, DTree);
  AllBalanceFactors(DelTree, Dummy);
  while BalTree<>nil do
  begin
```

```
    if BalTree^.BalFact in [Pl2, Min2] then
    begin
      if BalTree=DelTree then
      begin
        BalanceTree(BalTree);
        DelTree:=BalTree;
      end
      else
        BalanceTree(BalTree);
      AllBalanceFactors(DelTree, Dummy);
    end;
    BalTree:=BalTree^.Last;
  end;
end;
{*═══════════════════════════════════════════════*}
procedure DelAllTree(Var DelTree : AvlTree);
begin
  if DelTree<>nil then
  begin
    DelAllTree(DelTree^.Left);
    DelAllTree(DelTree^.Right);
    FreeMem(DelTree^.Data,DatSize);
    FreeMem(DelTree,KeySize+17);
    DelTree:=nil;
  end;
end;

end.
```

Listing 2: Demoprogramm

```
program AvlDemo;
{*********************************************}
{ Written in August '90 by P. Leibfried       }
{ (C) 1990 by Philipp Leibfried               }
{*********************************************}
uses Crt, AvlTrees;
{$V-}
type NamStr = String[30];
     AnyStr = String[80];
     AdrRec = record
                Name, Vorname, Stra, Nr, PLZ, Ort : NamStr;
              end;
var AdrFile : file of AdrRec; { Adressdatei    }
    AVLRoot : AvlTree;  { Baum, Hilfsknoten    }
    FSize   : Integer;  { Anzahl der Adressen  }
    Ch1     : char;
{*═══════════════════════════════════════════════*}
function FlExist(FileName: AnyStr): boolean;
var Fl : File;
begin
  Assign(Fl, FileName);
  FlExist:=false;
  {$I-}
    Reset(Fl);
    if IOResult<>0 then
    begin
      Close(Fl);
      if IOResult<>0 then
      begin
      end;
      Exit;
    end;
    FlExist:=true;
    Close(Fl);
    if IOResult<>0 then
    begin
    end;
  {$I+}
end;
{*═══════════════════════════════════════════════*}
procedure FileInfo;
begin
  GotoXY(5, 12);
  Write('Anzahl der Adressen:', FSize:5, '     ');
  Write('Hoehe des Baumes:', Height(AVLRoot):5);
end;
{*═══════════════════════════════════════════════*}
procedure DispAdress(Adress: AdrRec);
begin
  with Adress do
  begin
    GotoXY(3, 18);
    ClrEol;
    Writeln('Name  : ', Name);
    ClrEol;
    Writeln('  Vorname : ', VorName);
    ClrEol;
    Writeln('  Straße  : ', Stra);
    Writeln('  Nummer  : ', Nr);
    ClrEol;
    Writeln('  PLZ     : ', PLZ);
    ClrEol;
```

```
    Writeln('  Ort     : ', Ort);
  end;
end;
{*═══════════════════════════════════════════════*}
procedure SetUpScreen;
var I : Integer;
    B : array [1..2] of byte absolute WindMax;
begin
  B[1]:=80;
  B[2]:=25;
  TextBackGround(0);
  ClrScr;
  TextColor(7);
  GotoXY(5,4);
  Writeln('1) Adresse eingeben');
  Writeln('    2) Adresse suchen');
  Writeln('    3) Adresse löschen');
  Writeln('    4) Alle Adressen anzeigen - Sortierreihenfolge vorwärts');
  Writeln('    5) Alle Adressen anzeigen - Sortierreihenfolge
rückwärts');
  Writeln('    6) Programm-Ende');
  GotoXY(24,2);
  Write('AVL-Bäume / Demonstrationsprogramm');
  for I:=3 to 78 do
  begin
    GotoXY(I,11);
    Write('—');
    GotoXY(I,13);
    Write('—');
  end;
end;
{*═══════════════════════════════════════════════*}
procedure InitTree;
var RecPos   : Integer;{ Aktuelle Leseposition }
    NewAdress: AdrRec; { Eingelesene Adresse   }
    NKey     : NamStr; { Schlüssel             }
    Inserted : boolean;{ Prüfvariable für      }
                       { InsertNew             }
begin
  KeySize:=11; { Größe des Schlüssels          }
  KeyType:=KString;
  DatSize:=SizeOf(Integer);
  { Größe der Daten = 2 (für Datensatznummer)  }
  AVLRoot:=NIL; { Der eigentliche Baum         }
  Assign(AdrFile, 'ADRESSEN.DMO');
  RecPos:=0;
  if FlExist('ADRESSEN.DMO') then
  begin
    Reset(AdrFile);
    FSize:=0;
    while not EOF(AdrFile) do
    begin
      { Adresse einlesen,                      }
      { Schlüssel erzeugen                     }
      Read(AdrFile,NewAdress);
      NKey:=Copy(NewAdress.Name, 1, 10);
      { Schlüssel + Datensatznummer im Baum    }
      { ablegen, falls Adresse mit gleichem    }
      { Schlüssel nicht schon vorhanden.       }
      InsertNew(NKey, RecPos, AVLRoot, Inserted);
      if Inserted then
      begin
        FSize:=Succ(RecPos);
        FileInfo;
```

```
        Inc(RecPos);
      end;
    end;
    FileInfo;
  end else
  begin
    ReWrite(AdrFile);
    FSize:=0;
    FileInfo;
  end;
end;
{*═══════════════════════════════════════════════*}
procedure InputAdress;
var InpAdr   : AdrRec; { Eingegebene Adresse   }
    NKey     : NamStr; { Schlüssel             }
    I        : Integer;
    Inserted : boolean;{ Prüfvariable          }
                       { für Insert            }
begin
  GotoXY(3,15);
  Write('Bitte neue Adresse eingeben ...');
  repeat
    { Adresse von Tastatur einlesen            }
    with InpAdr do
    begin
      GotoXY(1, 18);
      Write(' Name   : ');
      Readln(Name);
      if Name<>'' then
      begin
        Write(' Vorname : ');
        Readln(VorName);
        Write(' Straße : ');
        Readln(Stra);
        Write(' Nummer : ');
        Readln(Nr);
        Write(' PLZ   : ');
        Readln(Plz);
        Write(' Ort   : ');
        Readln(Ort);
        NKey:=Copy(Name, 1, 10);
        InsertNew(NKey, FSize, AVLRoot, Inserted);
        { Falls Adresse mit dem selben         }
        { Schlüssel noch nicht existiert,      }
        { in Baum oder Datei einfügen          }
        if Inserted then
        begin
          Seek(AdrFile, FSize);
          Write(AdrFile,InpAdr);
          Inc(FSize);
        end;
        FileInfo;
        for I:=15 to 23 do
        begin
          GotoXY(1, I);
          ClrEol;
        end;
      end;
    end;
  until InpAdr.Name='';
  for I:=15 to 23 do
  begin
    GotoXY(1, I);
    ClrEol;
```

```
  end;
end;
{*═══════════════════════════════════════════════*}
procedure SearchAdress;
var SName : NamStr;
    SNode : AVLTree;
    DAdr  : AdrRec;
    I     : Integer;
begin
  GotoXY(3, 15);
  Write('Bitte Namen eingeben ...');
  GotoXY(3, 18);
  Write('Name   : ');
  Readln(SName);
  GotoXY(3, 15);
  { Durchsuche Baum nach Schlüssel             }
  SName:=Copy(SName, 1, 10);
  SNode:=SearchNode(SName, I, AVLRoot);
  { Falls Adresse gefunden, Bildschirmausgabe  }
  if SNode<>NIL then
  begin
    Seek(AdrFile,I);
    Read(AdrFile,DAdr);
    DispAdress(DAdr);
    GotoXY(3, 15);
    Write('Beliebige Taste --> weiter');
  end
  else
    Write('Adresse nicht gefunden ! Beliebige Taste ...',#7);
  Ch1:=ReadKey;
  for I:=15 to 24 do
  begin
    GotoXY(1, I);
    ClrEol;
  end;
 end;
{*═══════════════════════════════════════════════*}
procedure DeleteAdress;
var SName      : NamStr;    { Suchschlüssel    }
    SNode      : AVLTree;   { Hilfsknoten      }
    ReplAdr    : AdrRec;    { Datensatz zum    }
                            { überschreiben    }
    FindPos, I : Integer;   { FindPos = Daten- }
                            { satz zum Löschen }
    Dummy      : boolean;
begin
  GotoXY(3, 15);
  Write('Bitte Namen eingeben ...');
  GotoXY(3,18);
  Write('Name  : ');
  Readln(SName);
  GotoXY(3,15);
  { Durchsuche Baum nach Knoten                }
  SName:=Copy(SName, 1, 10);
  SNode:=SearchNode(SName, FindPos, AVLRoot);
  if SNode=NIL then
    Write('Adresse nicht gefunden ! ' +
          'Beliebige Taste ...'+#7)
    else
    { Falls Adresse gefunden, lösche aus       }
    { Datei und Baum                           }
    begin
      { Gehe zum entsprechenden Datensatz      }
      { der Datei                              }
```

```
      Seek(AdrFile, FindPos);
      { 1.Fall: Dieser Datensatz ist           }
      {         letzter Datensatz d. Datei     }
      if FindPos=Pred(FSize) then
      begin
        { Schneide letzten Datensatz ab,       }
        { lösche Knoten aus Baum               }
        Seek(AdrFile, Pred(FSize));
        Truncate(AdrFile);
        DeleteNode(SName, AVLRoot, Dummy);
        Dec(FSize);
      end
      else
      { 2.Fall : Zu löschender Datensatz       }
      {          mitten i.d. Adressdatei       }
      begin
        { Letzten Datensatz in Variable        }
        { ReplAdr lesen, Datensatz             }
        { abschneiden und zu löschenden        }
        { Datensatz überschreiben              }
        Seek(AdrFile, Pred(FSize));
        Read(AdrFile, ReplAdr);
        Seek(AdrFile, Pred(FSize));
        Truncate(AdrFile);
        Dec(FSize);
        Seek(AdrFile, FindPos);
        Write(AdrFile, ReplAdr);
        { Entsprechenden Knoten löschen        }
        DeleteNode(SName, AVLRoot, Dummy);
        { Im Knoten des neu positionierten     }
        { Datensatzes stehende Satznummer      }
        { aktualisieren                        }
        SName:=Copy(ReplAdr.Name, 1, 10);
        ReplaceData(SName, FindPos, AVLRoot, Dummy);
      end;
      Writeln('Adresse gefunden & gelöscht !');
      Write('  Beliebige Taste --> weiter ...');
      FileInfo;
    end;
    Ch1:=ReadKey;
    for I:=15 to 24 do
    begin
      GotoXY(1, I);
      ClrEol;
    end;
end;
{*═══════════════════════════════════════════════*}
procedure Show_in_Order(Forw : byte);
var LookNode : AvlTree;
    ActNum,I : Integer;
    DispAdr  : AdrRec;
begin
  if Forw=1 then
    LookNode:=Lowest(ActNum, AVLRoot)
  else
    LookNode:=Highest(ActNum, AVLRoot);
  GotoXY(3, 15);
  Write('Taste --> nächste Adresse ...');
  while LookNode<>NIL do
  begin
    Seek(AdrFile, ActNum);
    Read(AdrFile, DispAdr);
    DispAdress(DispAdr);
    Ch1:=ReadKey;
```

```
    if Forw=1 then
      LookNode:=NextNode(ActNum, LookNode)
    else LookNode:=LastNode(ActNum, LookNode);
  end;
  for I:=15 to 24 do
  begin
    GotoXY(2, I);
    ClrEol;
  end;
end;
{**********H A U P T P R 0 G R A M M***********}
begin
  SetupScreen;
  InitTree;
  repeat
    Ch1:=ReadKey;
    case Ch1 of
      '1' : InputAdress;
      '2' : SearchAdress;
      '3' : DeleteAdress;
      '4' : Show_in_Order(1);
      '5' : Show_in_Order(0);
    else
      begin
      end;
    end;
  until Ch1='6';
  { Ende : Datei schliepen - Baum löschen }
  Close(AdrFile);
  DelAllTree(AVLRoot);
  ClrScr;
end.
```