

AlgoDat — Ausarbeitung zum Übungsblatt 4

YRUCREM

Montag, 13. Mai 2002

VERSION 0.10 (B130502)

DANCINGCOMET korrigierte einen Tippfehler in Aufgabe 9.
SERSERI bemerkte, dass der Pseudocode für Aufgabe 6 etwas verkürzt werden konnte und WALTER HUBER machte mich darauf aufmerksam, dass ich beim verkürzen etwas zu viel weggenommen hatte.

Bei Aufgabe 7, habe ich Ab- statt Aufrundungszeichen benutzt und wurde von DOSE und ANTI BIT korrigiert.

Zum darstellen der Bäume wurde das Packet ‘‘qtree’’ verwendet. Die so erzeugten Bäume lassen sich leider nicht korrekt in einer pdf-Datei darstellen (die Linien zwischen den Knoten fehlen). Wer selbst L^AT_EX zur verfügung hat, sollte sich aus dem source .dvi oder .ps Dateien erstellen (die beiden png-Dateien und qtree.sty befinden sich in yrucrem.dr.ag/studium/algodat_1/<dateiname>).

Der L^AT_EX-Source, sowie die neueste Version dieses Dokuments sind unter yrucrem.dr.ag/studium/algodat_1/uebungsblatt_4.html erhältlich.

Aufgabe 1

Nein, der Algorithmus würde nicht als binäre Suche funktionieren.

Hier ein paar Zahlenfolgen, bei denen der Algorithmus nichts findet (die Zahlen die nicht gefunden werden können sind unterstrichen):

0, 1, 2, 3, 4, 5
0, 1, 2, 3, 4, 5, 6
0, 1, 2, 3, 4, 5, 6, 7, 8

Wenn wir beim letzten Beispiel die 5 suchen wollen, kommen wir sogar in eine Endlosschleife: der Algorithmus nimmt als m zuerst 4, das ist kleiner als 5, also zählt er die Hälfte von vier dazu und wir erhalten 6. Das ist größer als 5, also halbieren wir und kommen auf 3. Das ist wieder zu klein und die Hälfte von 3 wird zu 3 addiert und dann abgerundet, das ergibt 4! Also beginnt das Ganze wieder von Vorne.

Was außerdem bei diesem Beispiel passiert, ist, dass wir wieder auf die 3 zurückfallen, obwohl bereits festgestellt wurde, dass die gesuchte Zahl größer als 4 ist. Die Idee hinter der binären Suche ist aber, festzustellen, ob das gesuchte Element links oder rechts des gerade betrachteten Elementes ist — und dann dort weiterzusuchen und die andere Hälfte nicht mehr zu beachten. Dieser Algorithmus kann wieder in Bereiche des Feldes zurückkommen, von denen man schon weiß, dass das gesuchte Element nicht dort sein kann.

Aufgabe 2

Die zwölf Zahlen nennen wir x_1, \dots, x_{12} . Weiters muss gelten:

$$x_1 < x_2 < x_3 < x_4 < x_5 < x_6 < x_7 < x_8 < x_9 < x_{10} < x_{11} < x_{12}$$

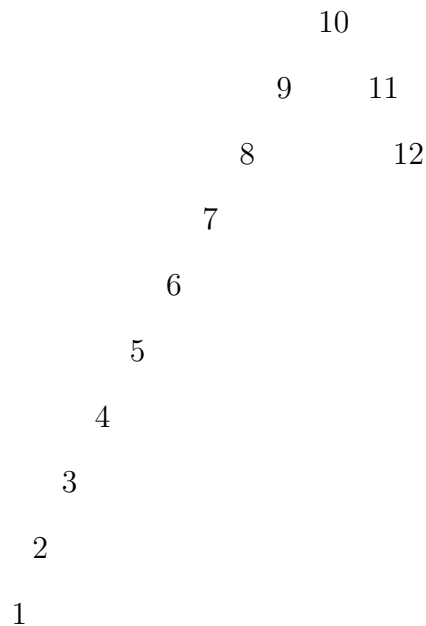
Es ist implementationsabhängig, ob man gleichgroße Elemente in den linken oder den rechten Unterbaum gibt, also habe ich nur *kleiner*-Zeichen gewählt (wenn man weiß wohin man sie gibt, kann man ein paar ' $<$ ' durch ' \leq ' ersetzen).

Wenn wir jetzt einen Baum machen wollen, der die Tiefe 9 hat und bei dem jedes rechte Kind wieder nur rechte Kinder hat — das gleiche für die linke

Seite — dann müssen wir auf die eine Seite zwei Elemente bringen und auf die Andere die restlichen Neun.

Wenn wir auf der linken Seite neun Elemente haben wollen, müssen wir zuerst das drittgrößte Element einfügen. Danach kommen die kleineren und die größeren, wobei nur wichtig ist, dass die kleineren Elemente *untereinander* in *absteigender* Reihenfolge und die größeren *untereinander* in *aufsteigender* Reihenfolge eingegeben werden.

Dadurch würde folgender Baum entstehen (es wurden nur die Indizes geschrieben):



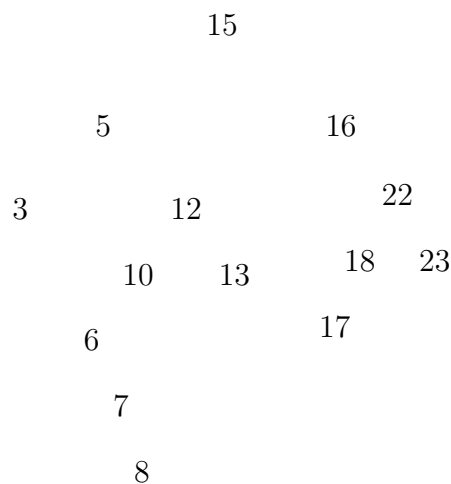
Dieser Baum würde zum Beispiel durch die Eingabe 10, 11, 12, 9, 8, 7, 6, 5, 4, 3, 2, 1 erzeugt werden, genauso würde aber 10, 9, 11, 8, 7, 6, 12, 5, 4, 3, 2, 1 funktionieren. Es ist wie gesagt nur wichtig, dass die kleineren Elemente *absteigend* und die größeren Elemente *aufsteigen* eingegeben werden.

Aufgabe 3

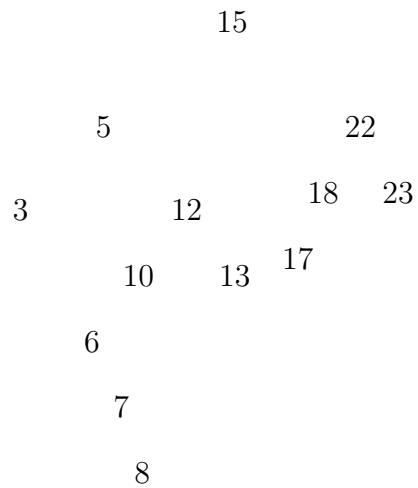
Wie im Skriptum beschrieben gibt es drei Möglichkeiten, wenn man einen Knoten aus einem natürlichen Suchbaum entfernen will:

1. Der Knoten hat keine Kinder: dann kann man ihn einfach löschen.
2. Der Knoten hat nur ein Kind: dann muss man ihn aus dem Baum „herausschneiden“ und sein Kind kommt an dessen Stelle.
3. Der Knoten hat zwei Kinder: dann muss man einen geeigneten Ersatzknoten finden, dazu eignen sich entweder der *Predecessor* oder – den wir auch nehmen werden – der *Successor*.

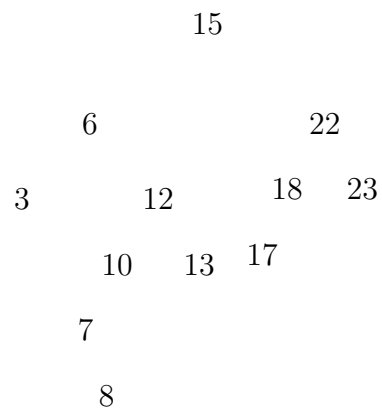
Zuerst sollen wir den Knoten 20 entfernen. Dieser hat zwei Kinder, also suchen wir uns seinen Successor (der kleinste Knoten im rechten Unterbaum von 20) und fügen ihn an seiner Stelle ein. Der Successor von 20 ist 22 und dieser Knoten hat keine Nachfolger, also kann 22 ohne weiteres an die Stelle von 20 gestellt werden. Danach sieht der Baum folgendermaßen aus:



Als nächstes soll die 16 entfernt werden. Dieser Knoten hat nur eine Kind, also können wir ihn einfach herauschneiden und sein einziges Kind an seine Stelle setzen. Danach sieht der Baum so aus:



Der Knoten den wir als nächstes entfernen sollen – die 5 – hat wieder zwei Kinder, also suchen wir den Successor von 5 und dieser ist der Knoten 6. Diesen Knoten schneiden wir aus dem Baum heraus – dadurch wird der Knoten 7 zum linken Kind von Knoten 10 – und fügen in an der Stelle von 5 ein. Das ist nun der Baum nach allen Operationen:



Aufgabe 4

Die Inorder Traversierung ruft sich zunächst rekursiv für den linken Unterbaum des gerade betrachteten Knotens auf, gibt dann den genannten Knoten aus und ruft sich dann selbst rekursiv für den rechten Teilbaum des Knotens auf. Wir starten bei der Wurzel, zuerst wird aber der Algorithmus gleich wieder für den linken Unterbaum aufgerufen, so kommen wir zum linken Kind der Wurzel. Und so geht es solange weiter bis wir das Blatt „ganz links unten“ erreicht haben. Das wird dann auch ausgegeben. Danach kommen wir wieder eine Rekursionsebene höher, geben diesen Knoten aus und bearbeiten sofort rekursiv den rechten Unterbaum dieses Knotens. So wird zuerst der gesamte linke Unterbaum bearbeitet, dann wird die Wurzel selbst ausgegeben und schließlich der gesamte rechte Unterbaum.

Die Preorder Traversierung gibt *sofort* den Knoten aus an dem sich der Algorithmus gerade befindet und bearbeitet dann erst seine Unterbäume (wieder zuerst den Linken dann den Rechten).

Wir wissen also jetzt, dass das erste Element der Preorder-Traversierung, die Wurzel des Baumes ist. Weiters wissen wir, dass in der Inorder-Traversierung alle Elemente die links von der Wurzel stehen in den linken Unterbaum gehören und alle die rechts stehen in den Rechten. Da liegt ein rekursiver Ansatz nahe. Für unser Beispiel sähe das so aus: 8 ist die Wurzel, 0, 1, 3, 5, 6, 7 sind im linken Unterbaum und 10, 11, 13, 17, 18 im Rechten:

8

0, 1, 3, 5, 6, 7 10, 11, 13, 17, 18

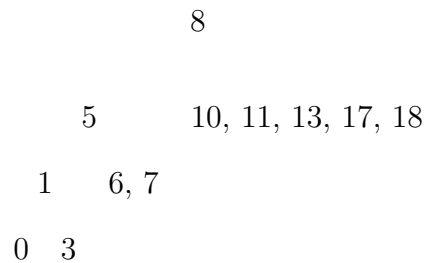
Wir gehen rekursiv zuerst für den linken Unterbaum vor: 5 ist dessen Wurzel, 0, 1, 3 sind links, 6, 7 sind rechts:

8

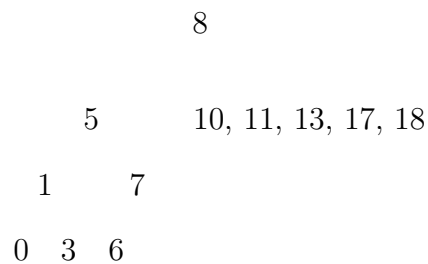
5 10, 11, 13, 17, 18

0, 1, 3 6, 7

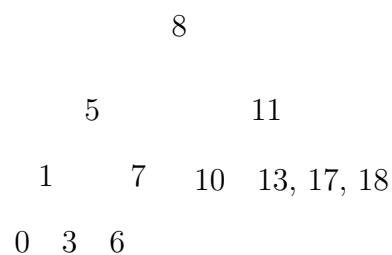
Wieder gehen wir rekursiv zum linken Unterbaum: 1 ist dessen Wurzel, 0 ist das linke Kind und 3 ist das rechte Kind:



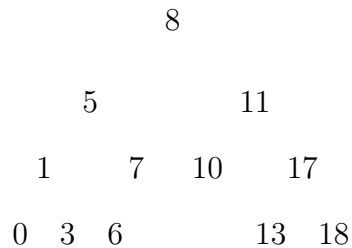
Damit ist ein kleiner Teilbaum bereits fertig. Wir gehen wieder hoch zur 5 und bearbeiten dessen rechten Unterbaum. Dessen Wurzel ist 7, das linke Kind ist 6. Einen rechten Unterbaum gibt es nicht:



Mit dem linken Unterbaum sind wir fertig. Die Wurzel des rechten Unterbaumes ist 11, links von der 11 ist nur die 10, rechts davon sind 13, 17, 18:



Für den linken Unterbaum gibt es nichts mehr zu tun, da ist nur noch ein Element, also gehen wir gleich zum rechten. Dessen Wurzel ist 17, links davon 13, rechts davon 18. Und damit wäre der Baum dann auch fertig rekonstruiert:



Das hier wäre der obige Algorithmus in Pseudocode. Die Variablen `In[]` und `Pre[]` sind global definiert (ich sehe keinen Sinn sie immer als Parameter zu übergeben) und enthalten die Ausgabe der In- und PreOrder-Traversierung. Die Variable `i` wird auf 0 initialisiert und zeigt auf das nächste Element, das wir aus `Pre[]` entnehmen werden und ist ebenfalls global definiert (ich wüsste zugegebenermaßen nicht, wie ich das anders machen sollte). Die Funktion `buildtree(l, r)` gibt einen Zeiger auf die Wurzel eines Unterbaumes zurück, die Rückgabe des ersten Aufrufs sollte also in einer Variablen gespeichert werden, die dann auf die Wurzel des gesamten Baumes zeigt. Die Parameter `l` und `r` geben die linke und rechte Grenze für das Feld `In[]` an. Die Funktion `find()` liefert die Position an dem der angegebene Wert im ebenfalls angegebenen Feld gefunden wurde. Hier nun der Pseudocode:

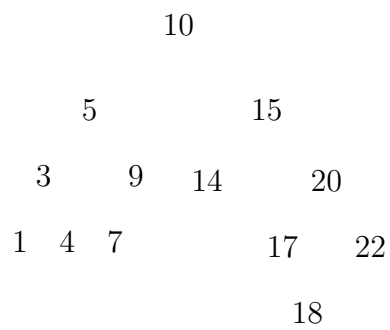
```

buildtree(l, r)
{
    root = NULL;
    if (l <= r) {
        root = Pre[i];
        pos = find(In, Pre[i])
        i = i + 1
        root.leftchild = build(l, pos - 1);
        root.rightchild = build(pos + 1, r);
    }
    return root;
}

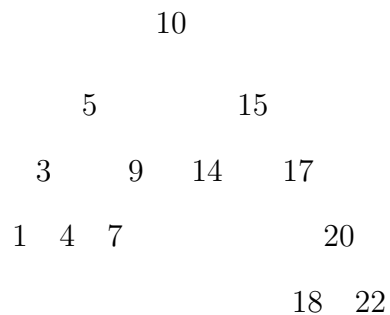
```

Aufgabe 5

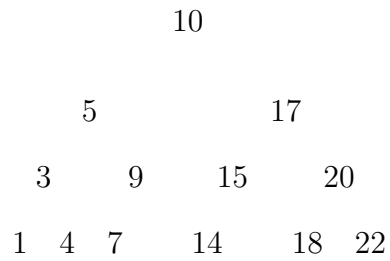
Wenn wir 18 einfügen ist es das rechte Kind von 17:



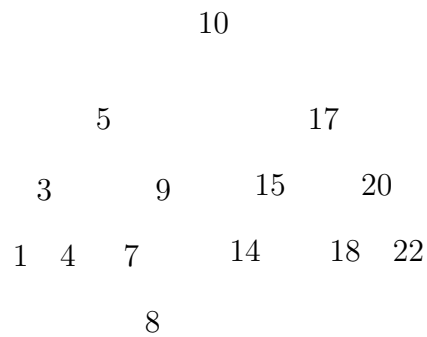
Die AVL-Bedingung ist im Knoten 15 verletzt, wobei zuerst der rechte und dann der linke Unterbaum tiefer ist, also müssen wir zweimal rotieren (zuerst rechts dann links). Wir rotieren zuerst rechts im Knoten 20 (wobei 17 sein linkes Kind (18) an 20 abgibt):



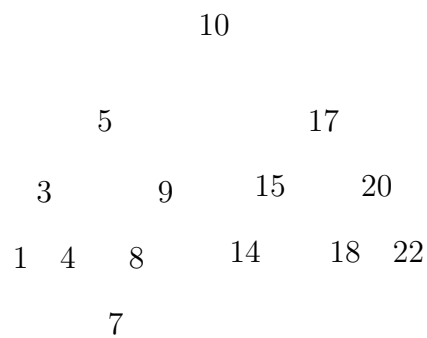
Dann rotieren wir in 15 nach links:



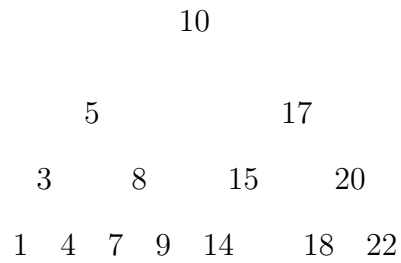
Nachdem die 8 eingefügt wurde ist diese das rechte Kind von 7:



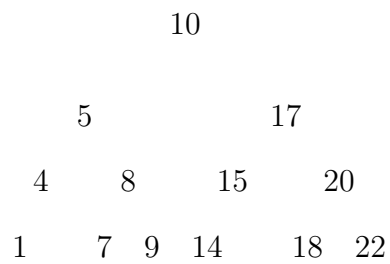
Die AVL-Bedingung ist im Knoten 9 verletzt und diesmal müssen wir zuerst links und dann rechts rotieren. Wir rotieren links in 7:



Dann rotieren wir rechts in 9:



Die 3 zu entfernen ist recht einfach. Wie beim natürlichen Suchbaum suchen wir uns dessen Successor, das ist die 4. Diese hat kein Kind also können wir sie einfach an die Stelle von 3 hinschreiben. Die AVL-Bedingung wurde nicht verletzt, also müssen wir auch nichts weiter machen.



Aufgabe 6

Die Erklärung des Algorithmus befindet sich darunter. Der Befehl `continue` soll wieder an den Schleifenanfang springen (also sofort einen neuen Durchlauf der Schleife beginnen).

```
findkleinsteselementdasgroesseristalsx(root, x)
{
    curr = root;
    found = NULL;
    while (curr != NULL) {
        if (curr.key < x) {
            curr = curr.rightchild;
            continue;
        }
        if (curr.key > x) {
            found = curr;
            curr = curr.leftchild;
            continue;
        }
        return curr;
    }
    return found;
}
```

Wenn wir an einem beliebigen Knoten sind und dieser kleiner ist als x , dann sind dieser Knoten und sein gesamter linker Unterbaum nicht interessant für uns, weil wir da wieder nur kleinere Elemente finden werden. Also gehen wir in den rechten Unterbaum und springen mit `continue` an den Anfang der Schleife.

Wenn das Element größer ist, wird es gleich übernommen, denn es ist sicher kleiner als etwaige Elemente die vorher gefunden wurden (wir sind ja in einem binären Suchbaum).

Wenn das Element weder kleiner noch größer ist dann muss es gleich x sein. Und näher können wir nicht an x heran, also können wir das gefundene Element sofort zurückgeben.

Wenn wir beim Baum ganz unten angekommen sind bricht die while-Schleife ab und wir geben `found` zurück. Das zeigt entweder auf das kleinste Element das größer ist als `x`, oder falls nichts gefunden wurde auf `NULL` (darauf wurde es initialisiert).

Der Algorithmus soll in $O(h(n))$ liegen, wobei $h(n)$ die höhe des Baumes ist. Das ist bei diesem Algorithmus sicher der Fall, weil wir maximal von der Wurzel bis zum untersten Blatt gehen. Der Algorithmus geht niemals den Baum wieder nach oben und es gibt auch keine inneren Schleifen.

Aufgabe 7

Das Aussehen der Bäume tut mir leid, aber ich wollte nicht alles selbst zeichnen und mit `qtree` sehen die Bäume etwas komisch aus.

1-3 lassen sich noch völlig Problemlos einfügen:

[1 2 3]

Nach dem Einfügen von 4 ist der Knoten zu groß und wir müssen splitten. Dabei wird das $\lceil \frac{m}{2} \rceil$ -te Element (in Deutsch: das Element in der Mitte — wobei bei einer ungeraden Anzahl aufgerundet wird), aus diesem Knoten entfernt und in den Knoten eine Ebene höher gegeben. Der linke Teil des gesplitteten Knotens liegt jetzt „links unter“ dem Element, das wir gerade nach oben gebracht haben (der rechte Teil rechts davon). Das wird jetzt so lange wiederholt bis alles so weit nach oben geschoben wurde, dass keine übervollen Knoten mehr vorhanden sind.

Hier wird also gesplittet, der 2er wird nach oben gebracht und dadurch zur neuen Wurzel. Seine Kinder sind [1] und [3 4]:

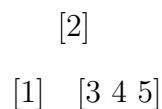
[2]

[1] [3 4]

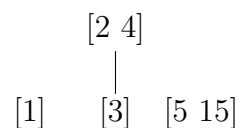
Beim einfügen beginnen wir bei der Wurzel und gehen solange nach rechts, bis

ein Element auftaucht das größer ist als, das Einzufügende. Wenn ein solches gefunden wird und *vor* diesem Element eine „Abzweigung“ zu einem Kind nach unten führt, gehen wir zu diesem Knoten hinunter und beginnen dort von vorne. Gibt es an dieser Stelle keine Kinder wird das Element einfach genau dort eingefügt. Wenn kein größeres Element vorhanden ist, geht man entweder zum ganz rechten Kind des Knotens oder falls keines da ist fügt man das Element gleich dort ein. Nach dem Einfügen muss dann überprüft werden, ob der Knoten auch nicht zu groß geworden ist.

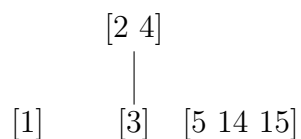
Hier wollen wir nun 5 einfügen. In der Wurzel ist kein Element, das größer ist als 5, aber wir haben ganz rechts außen ein Kind, also gehen wir dort hinunter. In diesem Knoten sind die Elemente 3 und 4, also wieder kein größeres Element und diesmal gibt es auch kein rechtes Kind. Also wir einfach nach dem 4er eingefügt:



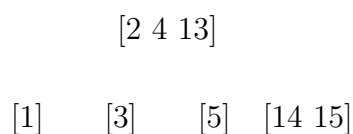
Ab jetzt gibt es nur mehr kurze erklärungen. Nach dem Einfügen von 15 müssen wir wieder splitten und die 4 wandert zur Wurzel hinauf:



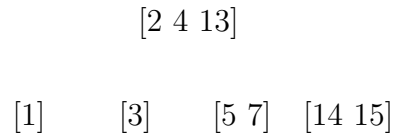
14 fügt sich zwischen 5 und 15 ein:



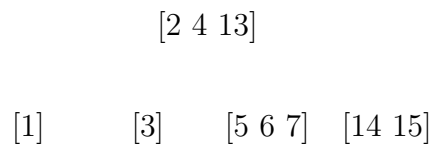
13 würde sich zwischen 5 und 14 einfügen, danach muss aber gesplittet werden und 13 wandert zur Wurzel:



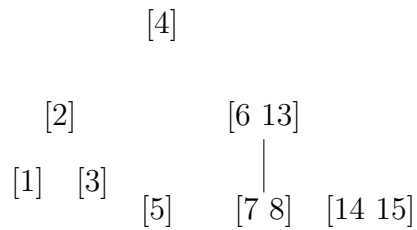
7 passt gleich nach 5:



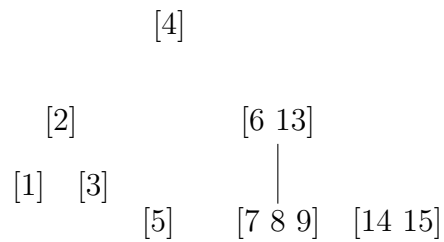
6 passt schön zwischen 5 und 7:



8 möchte in den gleichen Knoten wie 5, 6 und 7 dieser muss aber dann gesplittet werden und 6 wandert zur Wurzel. Diese ist dann auch zu groß und wird gesplittet:



9 lässt sich einfach einfügen:



Aufgabe 8

Zu dem Wert, der mit der ersten Hashfunktion ermittelt wird addiert man i -mal den Wert der zweiten Hashfunktion, wobei i ab 0 durchnummeriert wird. Beim ersten Versuch ein Element einzufügen wird also dieser Hashwert ermittelt: $h(k) = h_1(k) + 0 * h_2(k)$, was einfach nur dem ersten Hashwert entspricht. Ist diese Stelle frei, wird hier eingefügt, ist sie bereits besetzt, wird i um eins erhöht und man versucht es noch einmal (also eine Schleife).

Als erstes sollen wir die 5 einfüegen: $5 \bmod 13 = 5$. Diese Stelle ist noch frei, also wird hier eingefuegt.

Das naechste Element ist die 21: $21 \bmod 13 = 8$. Diese Stelle ist auch noch frei.

Als naechstes kommt 18: $18 \bmod 13 = 5$. Diese Stelle ist bereits besetzt, also kommt die zweite Hashfunktion auch zum tragen und wir erhalten:

$$18 \bmod 13 + 1 * ((7 + (18 \bmod 13)) \bmod 13) = 5 + ((7 + 18) \bmod 13) = 5 + 12 = 17$$

Damit man nicht über das Ende der Tabelle hinaus schreibt (wir haben keine Stelle 17 in der Tabelle) nimmt man das Ergebnis immer modulo der Anzahl der Stellen in der Tabelle (in unserem Fall 13). Dadurch beginnt man bei der Tabelle immer von vorne wenn man über das Ende hinaus ist (das macht man eigentlich immer, aber ich werde es nicht extra erwähnen, wenn das Ergebnis sowieso kleiner als 13 ist). $17 \bmod 13 = 4$ und diese Stelle ist noch frei.

Die weiteren Elemente gehe ich nicht mehr einzeln durch. Hier gleich die fertige Hashtabelle:

0	1	2	3	4	5	6	7	8	9	10	11	12
39			3	18	5	32	13	21	9			

Aufgabe 9

Diesesmal sollen die Hashtabelle mittels *linearem Sondieren* erstellt werden. Das ist ähnlich wie das Double Hashing — sogar einfacher. Wieder wird zuerst ein Hashwert ermittelt, und wenn dieser bereits belegt ist, sieht man einfach im darauffolgendem Feld nach, solange bis man eine leere Stelle findet.

Hier gleich die fertige Hashtabelle:

0	1	2	3	4	5	6	7	8	9	10	11	12
39	13		3		5	18	32	21	9			

Wenn man aus so einem Hashtable einen Eintrag löschen will, ist das etwas komplizierter, als man zuerst vielleicht denkt. Angenommen wir haben zwei Datensätze X und Y deren Hashwert der gleiche ist (zum Beispiel 5). Wenn wir zuerst X einfügen, ist X an der Stelle 5 (angenommen die Tabelle war vorher leer). Wollen wir nun Y einfügen haben wir eine Kollision und Y wird eine Stelle dahinter gespeichert. Wenn wir nun das Element X einfach löschen und später nach dem Element Y suchen passiert folgendes: der zu Y gehörige Hashwert ist 5, an dieser Stelle steht kein Eintrag, also würde man annehmen, dass das Element nicht im Hashtable vorhanden ist.

Deshalb werden Elemente nicht richtig aus einem Hashtable gelöscht, sondern, als gelöscht markiert. Wird nun ein neues Element eingefügt, dessen Hashwert genau dem des gelöschten Elements entspricht, wird das – sowieso als gelöscht markierte Element – einfach überschrieben.

Wird ein Element gesucht, bedeutet *als gelöscht markiert*, hier *war* einmal etwas und der Algorithmus weiß, dass er weitersuchen muss (wäre er auf ein vollkommen leeres Feld gestoßen, hätte er die Suche sofort abgebrochen).

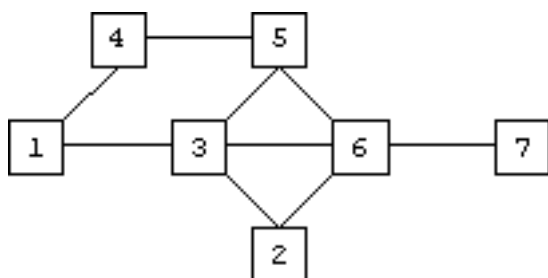
Nachdem 18, 39 und 5 als gelöscht markiert und 2, 1, 14 eingefügt wurden:

0	1	2	3	4	5	6	7	8	9	10	11	12
<i>39</i>	13	2	3	1	14	<i>18</i>	32	21	9			

(Die *kursiven* Zahlen sind als gelöscht markiert und die **fetten** sind neu eingefügt)

Aufgabe 10

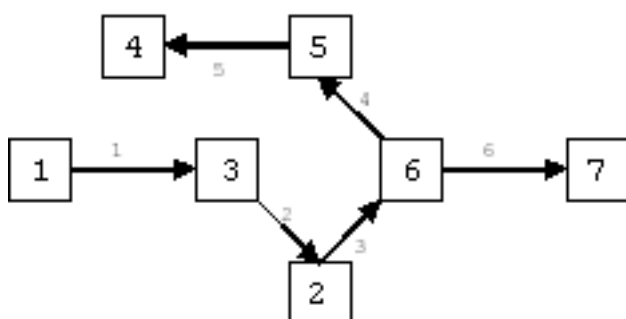
Ich denke jeder sollte aus einer Adjazenzliste (oder einer Matrix) den zugehörigen Graphen erstellen können, also ist hier gleich der vollständige Graph:



Depth-First-Search macht folgendes: es markiert den gerade besuchten Knoten als besucht (duh!) und ruft sich selbst rekursiv für jeden benachbarten Knoten auf, der noch nicht markiert ist. In dem unten gezeigten DFS-Baum wurden immer zuerst der benachbarte Knoten mit dem kleinsten Schlüssel gewählt.

Der kleinste, nicht markierte Knoten neben 1 ist 3. Von 3 geht's zu 2, dann zu 6, 5, 4. Da gibt es jetzt keine Knoten mehr die noch nicht markiert sind, also geht es wieder ein paar Rekursionsstufen hinauf — bis zur 6. Von dort kann man nämlich noch zum Knoten 7 und dann sind alle markiert.

Hier ist der DFS-Baum:



Die Schlechte Qualität der Graphiken tut mir leid. Ich hoffe ich finde bald eine andere Möglichkeit Bilder einzubinden.