Algorithmen und Datenstrukturen – Studentenfragen

Die nachfolgenden Fragen wurden von einen Ihrer Mitstudenten zusammengestellt um Unklarheiten vor dem ersten Test und dem AVL Implementierungsbeispiel auszuräumen. Ich werde versuchen diese so gut es geht zu beantworten:

Zum Lehrstoff:

1. *Wie bestimme ich die atomaren Operationen (z.B. 6 + 5n)?*
   Genau genommen müsste man an dieser Stelle tief in die Hardware Architektur eintauchen und bis auf das Assemblerniveau absteigen. Das macht aber bei der Analyse von Algorithmen keiner – aus offensichtlichen Gründen.
   Was man als atomar bezeichnet ist abhängig vom Beispiel. Für die Abschätzung sucht man sich bedeutende Operationen im zu beurteilenden Code (z.B. ein (Schlüssel-)Vergleich, eine Zuweisung, eine Addition / Multiplikation, Erstellen eines neuen Knotens, usw.). Diese summiert man unter Berücksichtigung der Programstruktur (Alternativen und Schleifen) auf und erhält das Ergebnis – wobei man die variable Problemgröße meist mit *n* einsetzt.
   Gegeben falls schreibt man auch unterschiedliche Operationen einzeln an (z.B. 3 Additionen + 2 Multiplikationen).
   In der Musterlösung zu Blatt 1 – Aufgabe 1) werden z.B. ein Vergleich und 2 Zuweisungen für das Einfügen gezählt – unter der Annahme das diese gleich „teuer“ sind, ergibt sich eine Anzahl von 3.
   Generell: der Begriff „atomare Operation“ ist in diesem Zusammenhang sehr schwammig – in solchen Fällen ist es immer günstig das man seien Wahl der atomaren Operationen bei Tests begründet.

2. *S. 43.: Induktionsbeweis allgemein & wie komme ich auf diese Werte bzw. Zahlen?*
   Mittels Induktionsbeweis kann man Eigenschaften über eine rekursiv (=induktiv) definierte Menge nachweisen.
   Was ist das? – nun, die Definition 1 ist eine natürliche Zahl und wann auch immer x eine natürliche Zahl ist, ist auch x+1 eine natürliche Zahl definiert z.B. alle natürlichen Zahlen. Man bestimmt eine oder mehrere Grundelemente (auch Atome genannt) – hier die *1* – und gibt eine oder mehrere *Produktionsregeln* an wie man zu weiteren Elementen kommt (hier *x => x+1*). Dasselbe kann man auch über Listen und Bäumen machen. (Z.B.: die leere Liste [] ist eine Liste. Sei *x* ein Element und R eine Liste, so ist x::R auch eine Liste – wobei „::“ das aneinanderhängen darstellt, daher 1::[2,3]=[1,2,3].)
   Mittels Induktionsbeweis kann man nun Eigenschaften für alle Elemente der induktiv definierten Menge (z.B. die natürlichen Zahlen oder die Menge aller Listen) beweisen. Zuerst benötigt man eine Induktionsbehauptung (Induktionshypothese) – was im Prinzip nur das zu zeigende ist (also z.B. die Feststellung das die Summe von 1 bis n gleich n*(n+1)/2 ist). Als

nächstes definiert man die Menge über welche die Induktion laufen soll (die induktive Definition davon, im Beispiel also die natürlichen Zahlen). Dann zeigt man im Induktionsanfang (Basis-Fall) diese Eigenschaft für alle Grundelemente der Induktiven Menge (im Beispiel: 1 = 1*(1+1)/2). Sollten mehrere Grundelemente vorhanden sein, muss die Eigenschaft für alle gezeigt werden. Schlussendlich muss im Induktionsschritt die zu zeigende Eigenschaft auch für jede der Produktionsregeln zum erzeugen neuer Element nachgewiesen werden. Im Beispiel muss man also für die Regel „x => x+1“ zeigen das aus der Annahme *„sum (1..x)=x*(x+1)/2“* die Behauptung *„sum (1..x+1)=(x+1)*(x+2)/2“* folgt.

sum(1..x+1) = sum(1..x) + (x+1) =*= x(x+1)/2 + (x+1) = (x(x+1)+2x+2)/2 = (x^2+3x+2)/2 =(x+1)(x+2)/2

Dabei wird an der Stelle die mit * gekennzeichnet wurde die Induktionshypothese für das Element *x* verwendet (die rot markierten Teile). Im Allgemeinen darf die Induktionshypothese für alle Werte < x+1 verwendet werden, nur nicht für x+1 selbst.

Auf Folie 43 wird die Behauptung T(n) = O(n log(n)) bewiesen mit (T(n) = <siehe Folie>). Um das zu zeigen wissen wir von der Definition von O(..) das wir Konstanten *d* und *n0* benötigen sodass *T(n) <= d*(n*log(n))* für alle *n >= n0*. Um es einfach zu halten werden auf der Folie nur 2er Potenzen berücksichtigt. Daher muss man zeigen:

T(2^k) <= d‘(2^k*log(2^k)) = d*2^k*k*

Hierfür führt man einen Induktionsbeweis über *k* als natürliche Zahl. Im Beispiel wird der Schritt zuerst ausgeführt und der Anfang auf 1 gesetzt.

Beweisen ist insgesamt eine Kunst für sich – die Wahl der Methode, die Festlegung von Werten, usw. – und wie beim Programmieren hilft da leider nur üben, üben, üben …

Mein Tipp: Jeden Beweis den man in irgend welchen Skripten zum Lernen vorfindet bis ins Detail durcharbeiten um ihn zu verstehen – mit der Zeit entwickelt sich das notwendige Gespür.

3. *Mein Taschenrechner unterstützt nur log(zahl) = Ergebnis. Wie kann ich den Logarithmus von log_b(a) auf eine andere Weise berechnen?*

   log_b(x) = log(x)/log(b)

4. *Wie kann ich die asymptotische Laufzeit abschätzen (raten)? Nur einige Tipps.*

   Daumen-mal-Pi : Tiefe der verschachtelten Schleifen zählen! Jede Verschachtelung erhöht die Potenz um +1. Keine schleife => O(n^0) = O(1), 2 verschachtelte Schleifen meist O(n^2), usw. …
   Sollte dann noch eine Rekursion oder ähnliches drin sein, ist meist noch ein log(n) drin.

ACHTUNG: dies ist nur eine Hilfestellung und muss nicht stimmen. Es kommt sehr auf die Grenzen der Schleifen an. Die Behauptung stimmt nur falls die verschachtelten Schleifen alle über einen nicht konstanten Teilbereich von 0..n iterieren.

5. *Wie kann man die 3 Fälle des Master-Theorems allgemein beweisen?*
   Das Master-Theorem muss in der AD Übung eigentlich nur angewendet werden können. Hierzu ist es notwendig das Master-Theorem zu kennen und nachzuweisen in welcher der 3 Fälle man sich befindet. Um dies zu zeigen, muss man wiederum die Zugehörigkeit der Funktion f(n) zu einer der 3 Komplexitätsklassen nachweisen (durch finden der Konstanten n_0 und c – siehe z.B. Folie 30).

   Ein Beweis für das Theorem an sich kann z.B. in Abschnitt 4.4 von Cormen et al's „Introduction to Algorithms" detailliert nachgelesen werden.

6. *Warum wird beim Heap-Sort der 10-er mit dem 1-er getauscht (S. 75 – 115 of 720)?*
   Nach dem Erstellen des Heaps wird das Wurzel-Element (das größte Element) mit dem letzten Element im Array vertauscht. Wenn man sich den Heap als Array „eingebettet" hinschreibt, sieht man dass das letzte Element die 1 ist.

7. *Schleifeninvariante (Schema bzw. Aufstellung)?*
   Wie in Frage 2 angedeutet, gibt es für die meisten Beweise keinen Algorithmus um an die korrekte Lösung zu kommen. Hierfür benötigt es Kreativität.

   Hier ein Beantwortung einer früheren Frage:
   „Schleifeninvarianten für Schleifen sind leider keinesfalls eindeutig und es gibt auch keinen Algorithmus zum herleiten dieser (weshalb es eines "kreativen" Akts bedarf, wie beim Programmieren). Aber es gibt Schemas, wie man sich einer Lösung annähern kann. In beiden Fällen im Beispiel ist es vermutlich am einfachsten man beginnt mit dem was man zeigen will (kleinstes Element der Rest-Liste bzw. vollständig sortierte Liste) und man überlegt sich wie diese Eigenschaft schrittweise bei jeder Schleifeniteration angenähert wird.

   Beispiel:

```
int i=0;
int sum = 0;
while (i<10) {
   sum += i;
   i++;
}
```

   zu Zeigen: sum = summe(0-9);

   Wenn man sich das so anschaut kann man erkennen, dass die summe schrittweise aufgebaut wird - und zwar nicht irgendwie, sondern nach dem Grundsatz das "sum = summe(0..(i-1))" immer gilt => so findet man eine mögliche Schleifeninvariante (welche in diesem Fall auch stimmt).

Jedoch hilft die Information das "sum = summe(0..(i-1)" gilt nach der Schleife wenig um daraus zu schließen das "sum = summe(0..9)". Was fehlt ist die Tatsache das "i=10" nach der Schleife. Ein beliebter Trick ist deshalb, dass man "i" begrenzt. So gilt als Invariante auch das "i <= 10". Wenn man beides zusammen setzt (invariante: "sum = summe(0..(i-1)) && i <= 10") kann man nach der Schleife zeigen, dass:

"!(bedingung) && Schleifeninvariante" =
"!(i<10) && sum = summe(0..(i-1)) && i <= 10" =>
"!(i<10) && i <= 10 && sum = summe(0..(i-1))" =>
"i = 10 && sum = summe(0..(i-1))" =>
"sum = summe(0..9)"

Meistens benötigt's so eine kombinierte Invariante. Die beiden Invarianten der Schleifen im Beispiel lassen sich nach diesem Schema herleiten. Am besten Schritt für Schritt durcharbeiten und immer wieder überlegen was fehlen würde um zum richtigen Schluss zu kommen (wie eben z.B. ein i<=10).“

8. *S. 81.: Wie komme ich auf diese Werte bzw. Umformungen (Komplexität von Heap-Sort)?*
   Um es kurz zu sagen: durch Analyse des Algorithmus und durch Wissen über Bäume.

   Die Beobachtungen sind generell für binäre Bäume gültig. Die Laufzeiten ergeben sich aus dem Algorithmus (jeweils der erste Schritt). Der Rest ist geschicktes Umformen um auf eine leserlichere Obergrenze zu kommen. Das Umformen selbst ist wieder so eine Angelegenheit welche man nur durch üben erlernen kann. Einen einfachen Algorithmus gibt es dafür nicht.

   Tipp: so etwas könnte man versuchen sich auf einem Blatt Papier Schritt für Schritt nachzurechnen um es richtig zu verstehen.

9. *S. 90. – 92.: Average-Case Quick-Sort: wie komme ich auf diese Werte bzw. Umformungen (durch Abschätzungen, usw.)?*
   Wie schon bei den vorhergehenden Antworten: es gibt keinen Algorithmus um auf eine Liste von Aussagen zu kommen, welche für einen Beweis notwendig sind. Ebenso gibt es kein Patentrezept für die notwendigen Umformungen.

   Wichtig bei diesen Folien ist es, sich die Annahmen zu überlegen und zu verstehen. Durch schrittweises durchgehen des Beweises sollte die Grund-Idee klar werden und der Beweis nachvollzogen werden können.

10. *Bucket-Sort-Beispiel (S. 99 (331 of 720)): wie kommt man auf den10-er bzw. auf die nachfolgenden Werte? -à Siehe Anhang!*
    Siehe Folie 101 – die 2te Schleife zur Berechnung der Start-Indizes. Wenn man den Algorithmus durchspielt setzt man cnt_ind[8] auf 12-2 = 10.

11. *Induktionsbeweis für Quick-Select (S. 111)?*
    Ist zwar nicht wirklich eine Frage, aber ja. Der Induktionsbeweis hat nur begrenzt was mit

Quick-Select zu tun. Eigentlich wird nur die gegebene Ungleichung für alle natürlichen Zahlen bewiesen. Hierfür werden anstelle eines Basisfalls 10 Basisfälle (die Zahlen 1 – 9) verwendet und die Produktionsregel *n => n+10* (welches wiederum auf eine andere Weise die Menge der natürlichen Zahlen definiert). Dieser „Trick“ vereinfacht den Induktionsschritt.

12. *Die amortisierten Kosten sind jene Kosten, welche danach wieder durch Guthaben aufgelöst werden. Liege ich richtig?*

    Laut Definition sind die amortisierten Kosten a_i einer Operation o_i

    $a_i = t_i + (G_i - G_{i-1})$

    Daher: es sind die Kosten einer Operation zuzüglich der Veränderung des Guthabens. Durch die Zuteilung erhöhter Kosten an billige Operationen kann auf diese Weise „Kapital“ für teure Operationen angespart werden (da, falls $a_i > t_i$ sich das Guthaben erhöht), welches dann durch reduzierte Preise von teuren Operationen verbraucht wird ($a_i < t_i$ reduziert das Guthaben). Ziel ist es die Komplexitätsklasse aller Operationen möglichst gering zu halten, ohne jemals das Konto (G) zu überziehen ($G_0 <= G_m$ für alle m).

Zum AVL Baum:

1. *Anfangs werden 9 vorgegebene Werte eingefügt. Danach zufällig erzeugte. Welche muss ich nun auf Existenz überprüfen?*

   Alle.

2. *Mit welcher Methode soll man anfangen insert, contains, balGrow, usw.?*

   Die Methode contains sollte am einfachsten sein und dient mehr oder weniger um sich mit dem Baum vertraut zu machen. Danach würde ich empfehlen mit insert zu beginnen und Schritt für Schritt nach unten zu arbeiten. Zuerst nur mit einem einfachen Einfügen in einen Binärbaum (wie in der Vorlesung besprochen), dann erweitern um die Rotationen (rotateLeft / rotateRight) und am Ende das balancieren (balGrow). Auf dies Weise hat man immer eine Version welche ausgeführt und schrittweise getestet werden kann.

   Tipp: beschränken Sie sich nicht auf den beigefügten Test Case. Um Teile Ihrer Implementierung zu überprüfen, schreiben Sie eigene Test Cases.

3. *Was macht der Befehl assert in balGrow?*

   Kurz gesagt, eine Assertion erlaubt es Bedingungen zu formulieren, welche von Java

überprüft werden. Sollte die gegebene Bedingung (die Einschränkung für hdiff) nicht erfüllt sein, so wird das Programm mit einem Fehler abgebrochen. Somit ist nach dem assert sichergestellt das hdiff entweder -1 oder 1 ist. Sie können dieses Statement jedoch auch problemlos ignorieren / entfernen.

Hinweis: im Allgemeinen ist es guter Programmierstill derartige Vorausbedingungen durch Assertions abzusichern – jedoch sollten dies im Normalfall niemals verletzt werden (da dann das Programm crashed). Deshalb ignoriert die Java Laufzeitumgebung diese Anweisungen im Normalfall um schneller Programme abarbeiten zu können. Mit der Option -ea lassen sich diese Checks einschalten (z.B. java -ea TestCase)

4. *Muss man bei insert die balGrow-Methode aufrufen (um AVL-Eigenschaft herzustellen) oder separate Berechnungen machen?*

   Während des *inserts* muss *balGrow* unter einer bestimmten Bedingung aufgerufen werden.

5. *Muss man bei contains den Baum mittels einer Schleife durchwandern?*

   Man muss nicht. Es lässt sich auch recht elegant mittels einer Rekursion lösen. Irgendein Konstrukt welches erlaubt mehrere Schritte im Baum zu „wandern“ wird jedenfalls benötigt.

6. *Muss man bei insert bevor man einfügt einen neuen Knoten erzeugen?*

   Ja – siehe hierfür Beispielcode aus der Vorlesung.

7. *Was bedeutet: „Hint: this might require some cumbersome case distinction code”?*

   Das an dieser Stelle eine Reihe von verschachtelten if-Anweisungen einzubringen sind und man durch die daraus folgende, erhöhte Komplexität des Codes sich nicht verunsichern lassen soll. Durchbeißen.

8. *Welche Variablen darf ich alles verwenden (Variablen in class Node)?*

   Alle. Sollten zusätzliche Variablen interessant sein, können auch neue hinzugefügt werden. Ich denke jedoch nicht dass irgendwelche fehlen. Das vorgegebene Programm ist nur eine Hilfestellung und kann beliebig umgebaut werden sofern die Aufgabenstellung erfüllt wird.

Die gegebene AVL Aufgabenstellung ist zugegebener Maßen Zeitaufwendig. Jedoch ist der Umgang mit Bäumen/Datenstrukturen in Programmen ein essentiellster Punkt welcher durch Übung verinnerlicht werden soll. Zusätzlich wurden die in der Aufgabenstellung zu implementierenden Algorithmen detailliert in der Vorlesung besprochen. Einer Umsetzung sollte deshalb lediglich ein gewisser Zeitaufwand im Wege stehen.