FernUniversität in Hagen

Lösungsvorschläge zur Nachklausur „1661 Datenstrukturen I“

19.09.2009

Aufgabe 1

(a)

Wir substituieren zunächst N durch 2^n:

$f(N) = f(2^n) = f(2^{n-1}) + 1 = f(2^{n-2}) + 2 = ... = f(2^1) + n - 1 = n = \log(N)$

Die Rekursionsgleichung g läßt sich durch Umformung in ähnlicher Weise auflösen:

$$g(2^n) = 2g(2^{n \angle 1}) + 2^n$$

$$\Leftrightarrow \frac{g(2^n)}{2^n} = \frac{g(2^{n \angle 1})}{2^{n \angle 1}} + 1 = \ldots = n \quad \Leftrightarrow \quad g(N) = N \log(N)$$

Wendet man Gleichung h auf sich selbst an, so erhält man folgende, aus dem Kurstext bekannte Summe:

$h(2^n) = 2^n + h(2^{n-1}) = ... = 2^n + 2^{n-1} + ... + 1 = 2^{n+1} - 1$

Also gilt $h(N) \approx 2N$.

(b)

Entsprechend der Definition der O-Notation prüfen wir zur Beantwortung der Frage

$$f(n) \stackrel{?}{=} O(g(n))$$

jeweils, ob solche Konstanten c und n_0 existieren, dass gilt:

$$f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

(i)

$(n + 1)! \leq c \cdot n! \Leftrightarrow$

$(n + 1) \cdot n! \leq c \cdot n! \Leftrightarrow$

$c \geq n + 1$

Der Wert von c muß also immer größer sein als n+1, um die Ungleichung zu erfüllen. Deshalb ist es nicht möglich, eine entsprechende Konstante c anzugeben: $(n+1)! \notin O(n!)$.

(ii)

$n^{n+1} \leq c \cdot n^n \Leftrightarrow$

$n \cdot n^n \leq c \cdot n^n \Leftrightarrow$

$c \geq n$

Analog zu (i) können wir auch hier keine solche Konstante c angeben: $n^{n+1} \notin O(n^n)$.

(iii)

$(n + 1)^n \leq c \cdot n^n \Leftrightarrow$

$$\left(\frac{n+1}{n}\right)^n \leq c \Leftrightarrow$$

$\left(1+\frac{1}{n}\right)^n \le c$

Da $\left(1+\frac{1}{n}\right)^n$ streng monoton steigend gegen den Grenzwert *e* strebt, gilt dies für alle $c \ge e$ und beliebige $n \ge 1$. Also gilt: $(n+1)^n = O(n^n)$.

Aufgabe 2

(a)

```
1 public Node find(Elem k){
2    Node n = Head;
3    for(int i = maxHeight-1; i >= 0; i--)// Beginne mit der „obersten“ Verzeigerung
4       while(n.next[i] != Tail && n.next[i].key < k)
5          n = n.next[i]; // Folge Verzeigerung auf der aktuellen Ebene
6    if (n.next[i] != Tail && n.next[i].key == k)
7       return n.next[i];
8    else
9       return null;
10}
```

(b)

Zunächst kommentieren wir den Quellcode:

```
22public void insert(Elem key) {
23   Node p = Head;
24   Node[] refs = new Node[maxHeight];       // Speicherung der Vorgänger...
                                              // ... des einzufügenden Knotens
25   for (int k = 0; k > maxHeight; k++) {refs[k] = Head;}// initialisiere Vorgänger
26   for (int k = maxHeight-1; k>=0; k--){          // Beginne auf der obersten Ebene
27      while (p.next != Tail && p.next[k].key < key){
28         p = p.next[k];                          // Suche auf Ebene k
29         refs[k] = p;                            // Merke letzten Zeiger auf Ebene k
30      }
31      p = p.next[0];
32      if (p != Tail && p.key == key) return;  // Schlüssel bereits enthalten
33   }
34   int h = idhgt();                            // Bestimme zufällige Höhe
35   Node newElem = new Node(key, h);      // Neuen Knoten erzeugen
36   for (int k = 0; k<maxHeight && k<h; k++) {
37      newElem.next[k] = refs[k].next[k];     // Zeiger auf Nachfolger setzen
38      refs[k].next[k] = newElem;             // Zeiger der Vorgänger anpassen
39   }
40   noElems++;
41}
```

Der Einfügealgorithmus durchsucht die Skip-Liste zunächst nach dem einzufügenden Schlüssel *x*. Wird *x* gefunden, so terminiert der Algorithmus. Bei der Suche wird ausgehend vom Startknoten in der obersten Liste gesucht, bis der Schlüssel *y* des Nachfolgerknotens größer als *x* ist. In diesem Fall wird der Knoten markiert (durch Eintrag in das Array *refs*) und die Suche ausgehend vom aktuellen Knoten in der nächstniedrigeren Liste fortgesetzt. Befindet man sich schließlich in der untersten Liste und trifft auf einen Schlüssel $y > x$ im nächsten Knoten, so hat man die richtige Einfügeposition gefunden. Der aktuelle Knoten wird markiert. Nun wird ein neuer Knoten mit Schlüssel *x* und einer zufälligen Höhe *h* erzeugt

und als Vorgänger der Nachfolger aller markierten Knoten sowie als Nachfolger aller markierten Knoten in die Skip-Liste eingetragen.

In Zeile 25 wird bedauerlicherweise ein falscher Vergleichsoperator „>“ statt „<“ eingesetzt. Dadurch wird das Array *refs* nicht initialisiert. In Zeile 37 führt dies zu einem Fehler (NullPointerException). Aus diesem Grund akzeptieren wir als Lösung ebenfalls „*insert* funktioniert nicht“ (mit Begründung).

(c)

Die Funktion *idhgt*() erzeugt eine ganzzahlige Zufallszahl von 1 bis *maxHeight*. Das Besondere jedoch ist die Verteilung der Zahlenwerte: Mit einer Wahrscheinlichkeit von 0,5 = 1/2 wird sofort die Zahl 1 zurückgegeben, mit einer Wahrscheinlichkeit von $0{,}5 \cdot 0{,}5 = 1/4$ nach dem zweiten Schleifendurchlauf die Zahl 2, allgemein nach dem *i*-ten Schleifendurchlauf mit einer Wahrscheinlichkeit von $p_i = 1/2^i$ die Zahl *i* (für $i < maxHeight$). Mit einer Wahrscheinlichkeit von $p_{(maxHeight-1)} = 1/2^{(maxHeight-1)}$ sind weitere Durchläufe nötig, die wiederum mit sinkenden Wahrscheinlichkeiten auf die Zahlen 1, 2, ... *maxHeight*-1 abbilden.

Insgesamt läßt sich sagen, dass für jedes hinreichend groß gewähltes *maxHeight* jede Zahl $1 \leq i \leq maxHeight$ mit einer Wahrscheinlichkeit von etwas über $p_i = 1/2^i$ zurückgegeben wird.

Anmerkung: Es kann (theoretisch) passieren, dass *idhgt*() nicht terminiert (falls *random*() immer eine Zahl < 0,5 liefert).

(d)

Da die Höhe jedes Knotens (außer Start- und Endknoten) zufällig mittels *idhgt*() bestimmt wird, hat für $maxHeight \to \infty$ (wir dürfen den Wert ja beliebig groß wählen) etwa die Hälfte aller Knoten die Höhe 1, ein Viertel die Höhe 2, ein Achtel die Höhe 3 usw.

Da die Knotenhöhen zufällig vergeben werden, können wir o.B.d.A. davon ausgehen, dass wir eine aufsteigend sortierte Folge mittels *insert*() in eine anfänglich leere Skip-Liste einfügen. Aufgrund der Verteilung der Ergebnisse von *idhgt*() wird etwa jeder zweite Knoten die Höhe 1 haben, die anderen Knoten eine Höhe > 1. Also ist die zu erwartende Anzahl aufeinanderfolgender Knoten mit Höhe ≥ 1 gerade 1. Von allen Knoten der Höhe ≥ 2 hat die erste Hälfte die Höhe 2, die zweite eine Höhe > 2. Auch hier erwarteten wir also wieder 1 aufeinanderfolgenden Knoten der Höhe 2. Vor und hinter jedem dieser Knoten erwarten wir einen Knoten der Höhe 1. Also erwarten wir jeweils 3 aufeinanderfolgende Knoten der Höhe ≥ 2. Für Höhen ≥ 3 ergibt sich eine erwartete Sequenzlänge von 7. Dieses Schema läßt sich bis zur Höhe *maxHeight* fortsetzen, allgemein erwarten wir Sequenzen von Knoten der Höhe 1 bis *i*, $1 < i \leq maxHeight$, mit einer durchschnittlichen Länge von 2^i-1.

Anmerkung: Falls *maxHeight* „klein“ gewählt wird, so wächst die Länge der Ketten aufeinanderfolgender Knoten maximal gleicher Höhe gemäß der Werteverteilung von *idhgt*() an.

Ebenfalls akzeptierte Antwort: Aufgrund des Fehlers in Zeile 25 funktioniert die *insert* Methode nicht wie gewünscht. Daher akzeptieren wir auch die folgende Antwort: Die erwartete Anzahl Knoten der Höhe *i* beträgt immer 0, ebenso die Anzahl benachbarter Knoten einer Höhe ≤ *i*.

(e)

Das beobachtete Verteilungsmuster der Knotenhöhen erzeugt eine hierarchische Listen-Struktur. Auf der untersten Ebene (Höhe ≥ 1) befindet sich etwa die Hälfte aller Schlüssel, auf der nächsthöheren etwa ein Viertel und so weiter. Ferner sind auch die erwarteten Abstände (Sequenzlängen) zwischen Knoten gleicher Höhe exponentiell zur Höhe des Knotens (der „Ebene“).

Eine solche Struktur kennen wir vom vollständig gefüllten binären Suchbaum. Dort erlaubt diese Struktur die Suche in logarithmischer Zeit. Dies gilt nun erwartungsweise auch für die mit *insert*() erzeugte Skip-Liste: Bei der Suche beginnt man auf der „obersten Ebene“, also der Liste mit Höhe *maxHeight*. In einer Skip-Liste mit *n* Schlüsseln enthält die oberste Liste im Durchschnitt $n \cdot 1/2^{(maxHeight-1)}$ Schlüssel.

Mit jedem Wechsel auf die jeweils nächstniedrigere Ebene der Höhe *i* verdoppelt sich die zwar die Anzahl der insgesamt über diese Liste erreichbaren Schlüssel. Die Suche nach einem Schlüssel *x* ist jedoch auf jeder Ebene beidseitig begrenzt: Nach „unten“ bzw. links durch den Knoten, in dem man eine Ebene absteigt, nach „oben“ bzw, rechts durch den Knoten der letzten Ebene, dessen Schlüssel größer war als *x* (bzw. durch den Endknoten). Da jede Ebene den Suchraum in der darunterliegenden etwa halbiert, dürfen wir mit logarithmischer Laufzeit für *find*() rechnen.

Anmerkung: Dasselbe gilt damit auch für die erwartete Laufzeit von *insert*() und *delete*().

Ebenfalls akzeptierte Antwort: Da keine Elemente eingefügt werden können beträgt die Laufzeit von *find* immer O(1). Da keine Elemente eingefügt werden können, entsteht auch kein hierarchisches Verteilungsmuster und eine Beantwortung des ersten Teils der Frage ist nicht möglich.

(f)

Aufgrund der zufälligen Zuweisung der Höhe zu jedem einzelnen Knoten mittels *idhgt*() darf man davon ausgehen, dass die Höhen der zu löschenden Knoten statistisch unabhängig sind von den Schlüsseln der zu löschenden Knoten (warum es auch gleichgültig ist, ob zufällige oder in der Skip-Liste aufeinanderfolgende Schlüsselfolgen gelöscht werden). Daher folgt die Verteilung der Höhen der gelöschten Knoten erwartungsweise gerade der Verteilung der Höhen in der Skip-Liste und ändert diese durch das Löschen nicht. Die Eigenschaften, welche die logarithmische Laufzeit von *find*() ermöglichen, werden also nicht berührt.

Ebenfalls akzeptierte Antwort: Da die Datenstruktur stets leer bleibt, bleibt konsequenterweise die Struktur auch unter beliebigen Löschversuchen erhalten.

Aufgabe 3

(a)

CocktailSort sortiert das übergebene Array *A* aufsteigend. Die **while**-Schleife wird ausgeführt, solange Elemente vertauscht wurden, also solange *A* nicht aufsteigend sortiert ist. Die **while**-Schleife selbst besteht aus zwei Phasen. In der ersten Phase erfolgt ein aufsteigender Durchlauf durch *A*, wobei zwei jeweils benachbarte Elemente miteinander verglichen und durch Vertauschen lokal sortiert werden. Wurde in der ersten Phase kein Paar von Elementen vertauscht, so ist *A* bereits sortiert und der Algorithmus terminiert. Anderenfalls erfolgt in der zweiten Phase ein absteigender Durchlauf durch *A*. Dabei werden wiederum benachbarte Elemente verglichen und ggf. durch Vertauschen lokal sortiert.

(b)

Laufzeitkomplexität: Sei $n = length(A)$. Als Laufzeitkosten zählen wir der Einfachheit halber die Anzahl der Vergleiche. In jedem Durchlauf der while-Schleife wird das gesamte Array *A* zwei mal durchlaufen. Es werden also jeweils $2n$-2 Vergleiche durchgeführt. Offensichtlich wird die while-Schleife höchstens $n/2$-mal durchlaufen, denn spätestens dann ist *A* sortiert. Wir erhalten also $O(n/2 \cdot (2n\text{-}2)) = O(n^2)$ als Laufzeit im worst-case.

Die Speicherkomplexität ist offensichtlich $O(n)$, da direkt im Array *A* sortiert wird.

Da die Laufzeit von *CocktailSort* im worst case $O(n^2)$ beträgt, die ideale Laufzeit eines Schlüsselvergleichssortierverfahrens jedoch in $O(n \log n)$ liegt, ist CocktailSort *nicht optimal*. Bezüglich des Speicherbedarfs ist *CocktailSort* dagegen optimal.

(c)

CocktailSort…

1. ist ein *internes* Sortierverfahren,
2. sortiert *durch Vertauschen*,
3. hat *quadratische* Laufzeit,
4. sortiert *in situ*, d.h. im Array selbst, ohne weiteren Speicher zu benötigen,
5. ist ein Schlüsselsortierverfahren und damit *allgemein*.

(d)

Sei x das größte Element in A und $n = length(A)$. O.B.d.A. sei $x = a_i$ das einzige größte Element in A. In Phase 1 des ersten Durchlaufs der **while**-Schleife wird a_i auf jeden Fall gefunden. Nun werden die benachbarten Elemente $a_i = x$ und $a_{i+1} = y$ miteinander verglichen. Gilt $a_i > a_{i+1}$ (und das ist natürlich der Fall), so werden a_i und a_{i+1} miteinander vertauscht. In a_i steht nunmehr y und in $a_{i+1} = x$. Als nächstes werden $a_{i+1} = x$ und $a_{i+2} = z$ miteinander verglichen. Da $a_{i+1} = x$ maximal ist in A, werden a_{i+1} und a_{i+2} vertauscht und so weiter, bis x am Ende der 1. Phase an Position a_{n-1} , also dem letzten und gemäß der endgültigen Sortierung endgültigen Platz angelangt ist. In Phase 2 geschieht ähnliches, allerdings wandert hier das kleinste Element im ersten **while**-Durchlauf definitiv auf den ersten Platz.

Man kann leicht sehen, dass im k-ten **while**-Durchlauf das k-größte Element auf Platz a_{n-k} und das k-kleinste Element auf Platz a_{k-1} bewegt wird. Danach werden diese Elemente nicht mehr bewegt, sie haben ihre endgültigen Positionen erreicht.

(e)

Man kann leicht sehen, dass im k-ten **while**-Durchlauf das k-größte Element auf Platz a_{n-k} und das k-kleinste Element auf Platz a_{k-1} bewegt wird. Danach werden diese Elemente nicht mehr bewegt, sie haben ihre endgültigen Positionen erreicht.

Diesen Sachverhalt kann man ausnutzen, indem man im k-ten **while**-Durchlauf die k-1 ersten und letzten Elemente von A nicht mehr betrachtet. Man spart so im k-ten Durchlauf $2(k\text{-}1)$ Vergleiche ein.

Das beobachtete Verhalten beweist, dass man höchstens $n/2$ **while**-Durchläufe benötigt.

(f)

Für die Anzahl der Vergleiche ergibt sich nunmehr:

$$\sum_{k=1}^{n/2}(n-1)-2(k-1) \overset{m:=\frac{n}{2}}{=} \sum_{k=1}^{m} 2m-2k+1$$

$$= \sum_{k=1}^{m} 2m - \sum_{k=1}^{m} 2k + \sum_{k=1}^{m} 1 = 2m^2 - 2\frac{m(m+1)}{2} + m$$

$$= 2m^2 - m^2 - m + m = m^2 = \frac{n^2}{4}$$

Daher bleibt die worst-case-Laufzeit trotzdem in $O(n^2)$.

Die Speicherplatzkomplexität bleibt natürlich bei $O(n)$.

Aufgabe 4

Aus einem *preorder*-Durchlauf und einem *postorder*-Durchlauf kann man relativ einfach den Baum rekonstruieren.

Wir wissen, dass beim *preorder*-Durchlauf zunächst die Ausgabe des Wurzelknotens erfolgt und die Ausgabe des *postorder*-Durchlaufs mit dem Wurzelknoten endet. *C* ist also der Wurzelknoten des gesuchten Baums.

Nach dem *preorder*-Durchlauf muss *X* ein Sohn von *C* sein. *R* kann jetzt laut *preorder*-Durchlauf Sohn von *C* oder *X* sein. Wenn *R* aber Sohn von *C* wäre, müsste es im *postorder*-Durchlauf nach *X* aufgeführt sein. Dies ist nicht der Fall, damit ist *R* äußerster linker Sohn von *X*. Hätte *R* noch Söhne, wären diese links von *R* im *postorder*-Durchlauf aufgeführt. Da *R* aber der erste Knoten im *postorder*-Durchlauf ist, hat *R* keine Söhne und ist damit der am weitesten links befindliche Blattknoten des gesuchten allgemeinen Baums.

Da *R* als Blattknoten keine Söhne hat, *T* aber unterhalb von *X* liegen muss, ist *T* Nachbarknoten von *R* und Sohn von *X*.

Mit ähnlichen Argumenten lassen sich alle weiteren Knoten in den Ergebnisbaum eintragen.