# Datenstrukturen und Algorithmen: Übungsblatt #3

Abgabe am 3. Mai 2018

**Finn Hess (378104), Jan Knichel (377779), Paul Orschau (381085)**

27.4.2018

# Aufgabe 1

**Teil a)**
Notation: Sei $N_h$ die Anzahl innerer Knoten eines Baumes der Höhe $h$.
Sei Aussage $A(h)$ gegeben durch $N_h \leq 2^h - 1$ (Der Baum hat höchstens $2^h - 1$ innere Knoten).

Beweis der Aussage $A(h)$ per Induktion nach $h \in \mathbb{N}$:

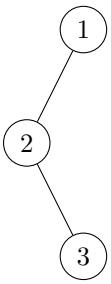
Induktionsanfang: $h_0 = 0$
$A(0)$ : Der Baum besteht nur aus der Wurzel, es gilt also $N_0 = 0 \leq 0 = 2^0 - 1$. ✓

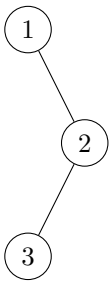
Induktionsvoraussetzung: Sei $h \in \mathbb{N}$ fest, aber beliebig, und es gelte $A(h)$.

Induktionsschritt: $h \to h + 1$
$A(h+1) : N_{h+1} \overset{*}{\leq} N_h + 2^h \overset{(I.V.)}{\leq} 2^h - 1 + 2^h = 2^h(1+1) - 1 = 2^{h+1} - 1$ ✓ □

*Der Baum der Höhe $h$ kann höchstens $2^h$ Blätter haben, die zu inneren Knoten des Baums der Höhe $h+1$ werden können. Daher gilt diese Ungleichung.

**Teil b)**
Preorder Linearisierung: 1 2 3
Postorder Linearisierung: 3 2 1
Zwei Bäume:




⇒ Nicht eindeutig!

**Teil c)**
Falls Mirror- und Inorder-Linearisierung eines Baumes gegeben sind, ist der Baum eindeutig bestimmt.

Beweis (konstruktiv):
Sei die Mirror-Linearisierung gegeben durch $M_1, M_2, \ldots, M_n$
und die Inorder-Linearisierung durch $I_1, I_2, \ldots, I_n$.
In der Mirror-Linearisierung tritt die Wurzel des Baumes stets als erstes auf, die Wurzel des Baumes ist stets $M_1$. Nun unterteilt dann die Inorder-Linearisierung in zwei Hälften:
$L_1, L_2, \ldots, L_{k-1}, M_1, L_{k+1}, \ldots, L_{n-1}, L_n$
Da es die Inoder-Linearisierung ist, gehören nun $L_1, \ldots, L_{k-1}$ zum **linken** Teilbaum und $L_{k+1}, \ldots, L_n$ zum **rechten** Teilbaum. Sollte es eine der beiden Knotensammlungen nicht geben, zum Beispiel weil $M_1$ an erster oder letzter Stelle auftritt, gibt es den jeweiligen Teilbaum einfach nicht.
Nun durchsucht man $M_2, \ldots, M_n$ nach dem ersten Element, welches auch in $L_1, \ldots, L_{k-1}$ vorkommt, also dem ersten Element, welches zum linken und nicht zum rechten Teilbaum gehört. Wir nennen dieses Element für den Moment $X$.
Nun ist gerade $M_2, \ldots$ bis (nicht einschließlich) $X$ die Preorder-Linearisierung des rechten Teilbaumes! Damit ist der rechte Teilbaum der Wurzel eindeutig bestimmt, denn wir kennen sowohl seine Inorder- als auch seine Preorder-Linearisierung.
Nun ist $X$ die Wurzel des linken Teilbaums, die verbleibenden Elemente der Mirror-Linearisierung dessen Mirror-Linearisierung und $L_1, L_2, \ldots$ bis (nicht einschließlich) $M_1$ dessen Inorder-Linearisierung. Nun kann man das Verfahren also von vorne beginnen, so lange bis keine Knoten mehr übrig sind und der Baum vollständig ist.

# Aufgabe 2

`traverse(G)` besucht jeden Knoten **mindestens** einmal, da jeder Knoten (außer der Wurzel) nach Eigenschaft 2 einen "Vorgängerknoten" besitzt, welcher die Funktion `visit(v)` auf ihm ausführen wird. Die Wurzel wird auch besucht, denn mit ihr startet die Rekursion in Zeile 8.
Jeder Knoten wird also mindestens einmal besucht, gibt sich aber **höchstens** einmal aus, da jeder Knoten sich "merkt", falls er schonmal besucht worden ist, und bei eventuellen weiteren Besuchen einfach gar nichts tut.
Damit wird jeder Knoten von `traverse(G)` also **genau einmal** ausgegeben!

# Aufgabe 3

**Teil a)**
Implementiert man die Liste mit zwei Zeigern, welche auf Anfang und Ende der Liste zeigen (bspw. doppelt verkettete Liste), so sind alle 5 Operationen in konstanter Zeit ausführbar.
Für `isEmpty` braucht man nur zu prüfen, ob beide Zeiger auf `Null` zeigen. Dafür spielt die Eingabelänge keine Rolle.
Für `enqueueFront` und `enqueueBack` fügt man einfach ein Element am Anfang/Ende der Liste hinzu und verschiebt den Zeiger. Die Dauer hängt ebenfalls nicht von der Eingabelänge ab.
Für `dequeueFront` und `dequeueBack` entfernt man das Element, auf welches der jeweilige Zeiger zeigt, und verschiebt den Zeiger danach. Sollte nur ein Element in der Liste sein, d.h. wenn nach der Operation die Liste leer ist, muss man zusätzlich noch beide Zeiger auf Null setzen. Die Dauer hängt wieder nicht von der Eingabelänge ab.

**Teil b)**
Es ist unmöglich, die Operationen in $\mathcal{O}(1)$ auszuführen, denn wenn man z.B. mit `enqueueFront` ein Element am Anfang der Liste einfügen möchte, müssen alle Elemente um 1 nach hinten verschoben werden, was nur in $\mathcal{O}(n)$ möglich ist, da es von der Länge der Liste abhängen muss. Das selbe gilt für das Entfernen eines Elements mit `dequeueFront`.

**Teil c)**

- `add` kann in $\mathcal{O}(n)$ ausgeführt werden, denn man muss das Element, welches hinzugefügt werden soll, im schlimmsten Fall mit allen bereits in $s$ enthaltenen Elementen vergleichen, benötigt also höchstens $n$ Vergleiche.
- `contains` kann in $\mathcal{O}(n)$ ausgeführt werden, denn um das gesuchte Element zu finden, muss man im schlimmsten Fall alle bereits in $s$ enthaltenen Elemente überprüfen, benötigt also höchstens $n$ Vergleiche.
- `union` kann nur in $\mathcal{O}(n)$ ausgeführt werden, falls die Sets (vielleicht mittels Hashwerten) so implementiert sind, dass `contains` in $\mathcal{O}(1)$ liegt. Dann erstellt man die Vereinigung von $s1$ und $s2$ indem man zunächst $s1$ kopiert und dann für jedes Element von $s2$ überprüft, ob es schon in $s1$ erhalten war. War es nicht enthalten, fügt man es zur Vereinigung hinzu. Da man also höchstens $n$-mal `contains` aufrufen muss (falls $s1$ keine Elemente enthält, also alle $n$ Elemente in $s1$ sind), liegt die Laufzeit von `union` in $\mathcal{O}(n)$.

**Teil d)**
Eine solche Implemetierung gibt es nicht, da die Erstellung des Ausgabesets auf jeden Fall von der Größe der beiden Eingabesets abhängen muss. Eine Vereinigung von zwei Sets mit jeweils einem Element braucht weniger Zeit zum Beschreiben des Speichers, Kopieren der Daten, etc. als eine Vereinigung von zwei Sets mit jeweils 1 Mio. Elementen.

# Aufgabe 4

**Teil a)**
$S(n) \in \mathcal{O}(1)$, wenn die doppelten Elemente direkt am Anfang der Liste stehen, da dann nur das erste Element abgespeichert wird, weil danach der Algorythmus abbricht.
$S(n) \in \Omega(n-1)$, wenn das doppelte Element am Ende der Liste steht, da dann die gesamte Liste bis auf das letzte (doppelte) Element abgespeichert wird.

**Teil b)**
Der Best Case ist 2, wenn die beiden 1er direkt am Anfang der Liste stehen, denn dann braucht der Algorithmus eine Zeiteinheit für die erste 1 und eine für die zweite und bricht danach ab, weil das doppelte Element bereits gefunden ist.
Der Worst Case ist n, wenn die zweite 1 an der letzten Stelle der Liste steht, da dann der Algorithmus jedes einzelne Element überprüfen muss, bis er am Ende das doppelte gefunden hat und dann abbricht. Der Average Case ist die Summe aller Wahrscheinlichkeiten, dass die zweite 1 an der Stelle i liegt multipliziert mit der entsprechenden Laufzeit:

$$
\begin{aligned}
A(n) &= \frac{\sum_{i=2}^{n} Anz\{\text{Eingaben mit zweiter 1 bei i}\} * t\{\text{Zweite 1 an der Stelle i}\}}{D_n} \\
&= \frac{\sum_{i=2}^{n} (i-1)(n-2)! * i}{\frac{n!}{2}} \\
&= \frac{2(n-2)!}{n!} \sum_{i=2}^{n} (i-1) * i \\
&= \frac{2(n-2)!}{n!} \sum_{i=2}^{n} [i^2 - i] \\
&= \frac{2(n-2)!}{n!} [\sum_{i=2}^{n} i^2 - \sum_{i=2}^{n} i] \\
&= \frac{2(n-2)!}{n!} [\frac{n(n+1)(2n+1)}{6} - 1 - \frac{3n(n+1)}{6} + 1] \\
&= \frac{2(n-2)!}{n!} * \frac{n(n+1)(2n+1) - 3n(n+1)}{6} \\
&= \frac{2(n-2)!}{n!} * \frac{2n^3 + 3n^2 + n - (3n^2 + 3n)}{6} \\
&= \frac{2(n-2)!}{n!} * \frac{2n^3 - 2n}{6} \\
&= \frac{2(n-2)!}{n!} * \frac{n^3 - n}{3} \\
&= \frac{2(n-2)!}{n!} * \frac{(n-1)n(n+1)}{3} \\
&= \frac{2}{3} * (n+1)
\end{aligned}
\tag{1}
$$

**Teil c)**

```
1 for (int i=0; i<list.size-1; i++) {
2   for (int j=i+1; j<list.size; j++) {
3     if (list[i] == list[j]) return true;
4   }
5 }
6 return false;
```

Der Algorithmus hat offensichtlich konstanten Speicherbedarf,
denn unabhängig von der Eingabelänge braucht er immer nur i und j als Variablen.
Die Worst-Case Laufzeit ist in $\mathcal{O}(n^2)$, denn die äußere Schleife läuft $(n-1)$-mal, die innere Schleife macht immer $(n-i)$ Vergleiche.

$$W(n) = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2-n}{2} \in \mathcal{O}(n^2) \tag{2}$$

**Teil d)**

```
1  int modus = 0;
2  int* anzahl = new int[n-1];
3
4  for (int i=0;i<n-1;i++) {
5    int x = l[i];
6    anzahl[x-1]++;
7  }
8
9  for (int i=0;i<n-1;i++) {
10   if (anzahl[i] > modus) {
11     modus = i;
12   }
13 }
14
15 delete[] anzahl;
16
17 return modus;
```

Der Algorithmus hat eine lineare Laufzeit, denn er hat keine verschachtelten `for`-Loops. Je nachdem, welche Zeilen also eine Zeiteinheit benötigen, beträgt die Laufzeit zum Beispiel $2n$.