

Hausaufgabe 3

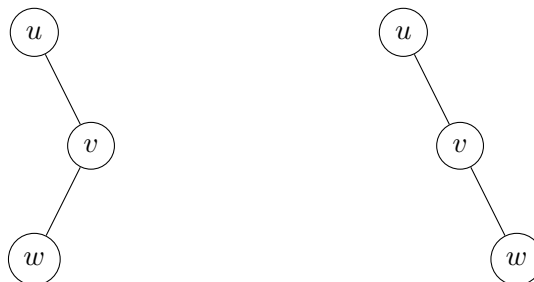
Aufgabe 1

a)

Es sei ein Binärbaum \mathcal{B} der Höhe h gegeben. Um eine maximale Anzahl an inneren Knoten zu enthalten, sollte er eine maximale Anzahl an Knoten enthalten, also vollständig sein. Damit enthält \mathcal{B} nach Skript $2^{h+1} - 1$ Knoten. Die Knoten in Ebene h sind nach alle Blätter von \mathcal{B} , alle anderen Knoten haben den maximalen out-degree von 2. Da Blätter keine inneren Knoten sind, ergibt sich also $(2^{h+1} - 1) - 2^h = 2^h - 1$ für die Anzahl der inneren Knoten von \mathcal{B} . \square

b)

Es sei ein Binärbaum \mathcal{B} gegeben sodass ein Knoten v einen out-degree von 1 hat und dieses Kind ein Blatt von \mathcal{B} ist. Wir nennen diesen Nachfolger hier mal w . Unabhängig davon, ob w ein linkes oder rechtes Kind ist, würde die Preorder-Traversierung (\dots, v, w, \dots) und die Postorder-Traversierung (\dots, w, v, \dots) lauten. Durch diese fehlende Information, ob ein gegebenes "Einzelkind" ein linkes oder rechtes ist, lässt sich der Baum nicht vollständig von den Traversierungen rekonstruieren. Wir geben ein kleines Beispiel:



Beide Binärbäume haben eine Preorder-Traversierung von (u, v, w) und eine Postorder-Traversierung von (w, v, u) . \square

c)

Es sei ein Binärbaum \mathcal{B} und dessen Inorder-Traversierung $i = (i_0, \dots, i_n)$ sowie Mirror-Linearisierung $l = (l_0, \dots, l_m)$ für $m, n \in \mathbb{N}$ gegeben. Nach Definition ist l_0 die Wurzel von \mathcal{B} .

Dann gibt es ein $k \in [0, n]$ mit $i_k = l_0$ sodass (i_0, \dots, i_{k-1}) den linken, und (i_{k+1}, \dots, i_n) den rechten Teilbaum der Wurzel darstellen. Weiter ist dann i_n der rechteste Knoten von \mathcal{B} . Es existiert weiter ein $j \in [0, m]$ mit $l_j = i_n$, sodass (l_1, \dots, l_j) die Preorder-Traversierung des rechten Teilbaumes von \mathcal{B} darstellt. Dieser Teilbaum ist wie bekannt dann eindeutig bestimmt. Dies lässt sich nun rekursiv fortführen indem wir l_{j+1} als Wurzel des linken Teilbaums betrachten und wie für \mathcal{B} vorgehen, insofern $j \neq n$, also \mathcal{B} überhaupt einen linken Teilbaum besitzt. Mit dieser Methode lässt sich ein gegebener Binärbaum eindeutig aus seiner Mirror-Linearisierung und Inorder-Traversierung rekonstruieren.

Aufgabe 2

Der gegebene Algorithmus ist in gewisser Weise ein Variante des bekannten Depth-First-Search. Wir besuchen jeden Knoten nur genau einmal, denn:

Sobald ein Knoten v besucht wird, werden alle von ihm aus erreichbaren Knoten besucht, also

$$M := \{v' \in V \mid v' \neq v \wedge \exists (v_0, v_1, \dots, v_n) : v_0 = v \wedge v_n = v' \wedge \forall i \in [1, n] : (v_{i-1}, v_i) \in E\}$$

Da der Algorithmus auf einem azyklischem Graphen arbeitet, gilt $v \notin M$, man kann also nicht wieder zu v zurückkommen, während man seine Nachfolger besucht. Sobald man dann alle Nachfolger $v' \in M$ besucht hat, wird durch eine Member-Variable angegeben dass v schon besucht wurde. Da ein Knoten beliebig viele Vorgänger bzw. eingehende Kanten haben kann, könnte man im weiteren Verlauf des Algorithmus nochmal bei v vorbeikommen. Jedoch wird zu Beginn von `visit(v)` überprüft, ob v schon besucht wurde. Somit wird jeder Knoten eines DAG von dem gegebenen Algorithmus genau einmal besucht.

Aufgabe 3

a)

Ja, alle 5 Operationen liegen in der Komplexitätsklasse $\mathcal{O}(1)$ (konstante Laufzeit, unabhängig von n).

`isEmpty()` überprüft in einem Schritt ob `first` leer ist.

`enqueueFront()` fügt ein Element als neues `first` Element ein und verkettet das ehemalige erste mit dem neuen.

`enqueueBack()` fügt ein Element als neues `last` Element ein und verkettet das ehemalige mit dem neuen.

`dequeueFront()` entfernt das derzeitige `first` Element und setzt `first` auf das ehemalige zweite der Schlange.

`dequeueBack()` ist analog zu `dequeueFront()`.

b)

Die Laufzeit der fünf Operationen auf einem unbeschränktem Array ist nicht konstant, da für den Befehl `dequeueFront()` zuerst das erste Element gelöscht werden muss und dann alle anderen Elemente verschoben werden müssen um die Umsetzung mit nur einem `last`-Zeiger zu ermöglichen.

c)

Die Operationen `add()` und `contains()` liegen in $\mathcal{O}(n)$ da Sie beide das ganze Set durchgehen müssen um zu überprüfen ob ein Element enthalten ist oder nicht.

Wobei `contains()` noch eine extra Zeiteinheit braucht um das Element hinzuzufügen, dennoch bleibt die Operation in $\mathcal{O}(n)$.

Ob die Aussage korrekt ist kommt auf die Funktionsweise von `union()` an, da die Frage ist, ob doppelte Element gelöscht werden müssen oder nicht.

Falls sie nicht gelöscht werden müssen bleibt auch `union()` in $\mathcal{O}(n)$. Ansonsten bräuchte die Operation n^2 Schritte und läge somit in $\mathcal{O}(n^2)$.

d)

Fallunterscheidung:

Fall 1: Doppelte Elemente müssen nicht gelöscht werden:

Durch Implementierung mit verketteten Listen die man einfach aneinanderhängt braucht die Operation `union()` nur eine Operation und liegt somit in $\mathcal{O}(1)$.

Fall 2: Doppelte Elemente müssen gelöscht werden:

In diesem Fall ist eine Implementierung in konstanter Laufzeit nicht möglich, da für jedes Element aus `set2`, `set1` ganz durchlaufen werden muss.

Aufgabe 4

a)

Die Speicherkomplexität ist in beiden Fällen gleich da das Array vorher schon mit $k+1$ Einträgen angelegt werden muss.

b)

$B(n,k) = 2$ (Beide Einsen sind direkt am Anfang hintereinander).

$W(n,k) = n$ (Die ganze Liste muss durchgegangen werden).

Die Anzahl an Permutationen der List ist $\frac{n!}{2}$. Die Anzahl an Permutationen der Liste sodass die 2te 1 an Index $i \in [0, n-1]$ steht, ist $i(n-2)!$. Da wir von einem Laplaceraum ausgehen ist jedes Ereignis gleichwahrscheinlich. Wir müssen die Liste immer bis zur 2ten 1 durchgehen, also $(i+1)$ Operationen durchführen. Es folgt:

$$A(n,k) = \frac{2}{n!} \sum_{i=1}^n (i+1)i(n-2)! = \frac{2}{n(n-1)} \sum_{i=1}^n i(i+1) = \frac{2(n+1)(n+2)}{3(n-1)} \approx \frac{2}{3}n$$

c)

```
1 int tmp, z;
2 for (i in l)
3     tmp = i
4     z = 0
5     for (j in l)
6         if (tmp == j)
7             z++
8     if (z > 1)
9         return true
10 return false
```

Dieser Algorithmus hat eine konstante Speicherkomplexität in dem Worst-Case, da er unabhängig von n immer nur zwei Werte speichern muss. Die Asymptotische Komplexität des Algorithmus im Worst-Case ist in $\mathcal{O}(n^2)$ (wenn nur die If-Abfragen als Operationen gelten).

d)

```
1  if (l.length <= 2)
2      return l.first
3
4  nums[] = 0^n
5
6  for (i in l)
7      nums[i]++
8
9  max = 0
10 maxNum = 0
11 current = 0
12
13 while (current < n)
14     if (nums[current] > maxNum)
15         maxNum = nums[current]
16         max = current
17
18     ++current
19
20 return max
```

Der Algorithmus hat eine lineare Laufzeit da er genau $2n$ If-Abfragen durchführt in jedem Fall außer `l.length <= 2` durchführt. Außerdem ist der Platzbedarf $n + 3 \leq n \cdot (\log n - 1) + 4 \log n$ in allen Fällen.

```
1  if (l.length <= 2)
2      return l.first
3
4  maxNum = 0
5  maxOccurrences = 0
6
7  for (i in l)
8      occurrences = 0
9
10     for (j in l)
11         if (i == j)
12             occurrences++
13
14     if (occurrences > m)
15         maxOccurrences = occurrences
16         maxNum = i
17
18 return maxNum
```

Der Speicherbedarf ist in allen Fällen konstant und somit kleiner $4 \log n < 5 \log n$.