

## Hausaufgabe 3

---

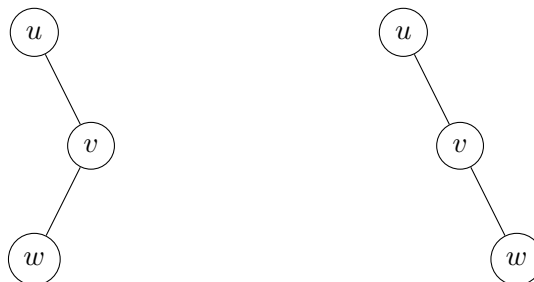
### Aufgabe 1

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)

## Aufgabe 2

Der gegebene Algorithmus ist in gewisser Weise ein Variante des bekannten Breadth-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 Element 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.