Gruppe **14**
Datenstrukturen und Algorithmen

Phil Pützstück, 377247
Benedikt Gerlach, 376944
Sebastian Hackenberg, 377550

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 $(\cdots, v, w, \cdots)$ und die Postorder-Traversierung $(\cdots, w, v, \cdots)$ lauten.

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

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

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 ehmalige erste mit dem neuen.

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

dequeueFront() enftfernt das derzeitige first Elemnt und setzt first auf das ehmalige zweite der Schlange.

dequeBack() analog zu dequeFront().

b)

Die Laufzeit der fünf Operationen auf einem unbeschränktem Array ist nicht konstant, da für den Befehl dequeFront() zuerst das erste Elemnt 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 dieses enthalten ist oder nicht.Wobei contains() noch eine exttra Zeiteinheit braucht um das Element hinzuzufügen, dennoch bleibt die Operation immernoch in $\mathcal{O}(n)$.
Ob die Aussage korrekt ist kommt auf die Funktiionsweise 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 $\mathcal{O}(n^2)$ und nicht mehr in $\mathcal{O}(n)$.

d)

Fallunterscheidung:

1.Fall Doppelte Elemente müssen nicht gelöscht werden:
Durch Implemtierung mit verketteten Listen die man einfach aneinanderhängt braucht die Operation union() nur eine Operation und liegt somit in der konstanten Komplexitätsklasse $\mathcal{O}(1)$.

2.Fall Doppelte Elemente müssen gelöscht werden:
In diesem Fall ist einen Implemtierung mit konstanter Laufzeit nicht möglich, da für jedes Element aus set2, set1 ganz durchlaufen werden muss.