Universität Karlsruhe
Fakultät für Informatik
ITI Wagner

2. Musterlösung

Problem 1: Least Common Ancestor 3pt

(a) Aus Zeilen 3 und 4 ist klar, dass LCA($wurzel[T]$) eine Folge von rekursiven Aufrufen LCA(u) entsprechend einer Preorder-Traversierung von T erzeugt. Somit findet die Schwarzfärbung in Zeile 7 für jeden Knoten u genau einmal, in Postorder-Reihenfolge, statt und induziert somit eine Ordnung der Knoten von T. Sei $\{u, v\} \in P$. Wenn nun im Aufruf LCA(u) Zeile 10 für $\{u, v\}$ ausgeführt wird, dann liegt Knoten v vor Knoten u in Postorder-Reihenfolge. Somit ist im Aufruf LCA(v) Knoten u noch nicht schwarz gefärbt, und Zeile 10 wird nicht ausgeführt. Somit folgt die Behauptung.

(b) *Hilfsbehauptung*: Nach Ausführung von Zeile 7 in LCA(u) beinhaltet die Menge FIND(u) alle Knoten die im Unterbaum $T(u)$ unterhalb von v liegen. Beweis per Induktion über die Höhe $h(v)$ von v.

- I.Anfang: Sei $h(v) = 0 \Rightarrow v$ ist Blatt $\Rightarrow$ Beh. gilt
- I.Annahme: Beh. gelte für alle Knoten u mit Höhe H
- I.Schluss: Sei $h(v) = H+1$. Für alle Kinder w_i von v gilt: $h(w_i) < h(v)$, und somit die Annahme. Während dem Aufruf LCA(v) werden die Mengen FIND(w_i) mit der Menge u vereint (Zeile 5). Somit gilt am Ende von LCA(v) die Behauptung.

Nun gilt die eigentliche Behauptung wie folgt:
Sei $D(u)$ die Anzahl Mengen in der Datenstruktur beim Aufruf LCA(u). Die rekursiven Aufrufe von LCA(u) erfolgen nach (a) in Preorder-Reihenfolge. Per Induktion über diese Folge ergibt sich:

- I.Anfang: Für LCA($wurzel[T]$) gilt die Behauptung offenbar.
- I.Annahme: $D(u)$ = Tiefe(u) für alle Knoten u *vor* Knoten v (bzgl. Preorder).
- I.Schluss: Beim Aufruf von LCA(v) gilt Tiefe(v) = Tiefe(Vater(v)) + 1. Fallunterscheidung:

Fall 1 Kein Geschwister von v liegt *vor* v: $D(v) = D(\text{Vater}(v)) + |\{\{\text{Vater}(v)\}\}| = D(\text{Vater}(v)) + 1 \Rightarrow$ Beh. gilt.

Fall 2 Seien $\{w_1, \ldots, w_k\}$ Geschwister von v die *vor* v liegen. $\forall w_i$: wegen Hilfsbehauptung enthält die Menge FIND(w_i) den Unterbaum von w_i, und in Zeile 5 von LCA(v) wird FIND(w_i) mit FIND(Vater(v)) vereint. Somit gilt auch hier $D(v) = D(\text{Vater}(v)) + 1$.

(c) Entsprechend (a) sei O.B.d.A. v *vor* u in der Postorder Reihenfolge (Folge der Schwarzfärbung). Also ist v beim Aufruf von Zeile 8 in LCA(u) bereits schwarz. Der Algorithmus gibt Vorfahr[FIND(v)] aus. Zu prüfen ist, ob dies der korrekte LCA ist. Zunächst ist durch die Zeilen 5 und 6 klar, das für jeden Knoten w stets gilt: Vorfahr[FIND(w)] ist ein Vofahr von w, falls definiert, was für u und v am Ende von LCA(u) gewährleistet ist.

Fall 1 v sei im Unterbaum von u. Nach Hilfsbehauptung gilt: $v \in$ FIND(u) somit gilt FIND(v) = FIND(u). Weiterhin gilt, dass wegen Zeile 6 in LCA(u) gilt Vorfahr[FIND(u)] = u. Somit ist Vorfahr[FIND(v)] = Vorfahr[FIND(u)] = u, was offenbar der korrekte LCA ist.

Fall 2 v sei nicht im Unterbaum u. Sei q der korrekte LCA von v und u, und sei der Pfad $\Pi = (u = w_1, \ldots w_k = wurzel[T])$ der Pfad von u zur Wurzel von T, dann gilt $q \in \Pi$. Zu zeigen ist: Vorfahr[FIND(v)] = q.

- Angenommen Vorfahr[FIND(v)] = x ist echter Vorfahr von $q \Rightarrow u$ und v sind im selben

Unterbaum von $x \Rightarrow$ wegen Vorfahr[FIND(v)] $= x$ wurde in LCA(x) Zeile 5 bereits für den Unterbaum von x, der v und u enthält durchgeführt $\Rightarrow$ LCA(u) ist schon abgearbeitet $\Rightarrow$ Widerspruch.

- Angenommen Vorfahr[FIND(v)] $= x$ ist echter Nachkomme von q und Element von $\Pi \Rightarrow$ x ist kein Vorfahr von $v \Rightarrow$ Widerspruch.

(d) Wir gehen von der in der Vorlesung erarbeiteten Version der UNION-FIND Datenstruktur aus. Wir betrachten eine ganzheitliche amortisierter Analyse über alle Rekursionen:

- Zeilen 2 und 7 werden n mal ausgeführt $\Rightarrow O(n)$
- Zeilen 1,5,6 sind $\Theta(n)$ Operation aus MAKESET, UNION, FIND $\Rightarrow O(nG(n))$
- Falls $|P| \in \Omega(n^2)$, werden Zeile 9 und 10 $\Omega(n^2)$ mal ausgeführt $\Rightarrow n^2$ FIND Operationen

Insgesamt wird die Laufzeit also majorisiert von n^2 Operationen vom Typ MAKESET, UNION, FIND, somit ist eine gute obere Schranke für die Laufzeit von LCA $O(n^2G(n))$.

Anmerkung: Da lediglich die Anzahl der FIND Operationen quadratisch in n ist, könnte man bei dieser Anwendung auch ein einfaches Array als Datenstruktur nutzen, dessen Einträge die enthaltende Menge eines Elements angeben. Jeweils in $O(1)$ können somit MAKESET und FIND durchgeführt werden, und UNION in $O(n)$, was einen amortisierten Gesamtaufwand von $O(n^2)$ für LCA zur Folge hätte.

Problem 2: Nachbarschaftstratsch 2pt

(a) Folgender Algorithmus löst das Problem.

Algorithmus 1 : TRATSCHREICHWEITE

Eingabe : Menge M, Array D, Wert $D_{\max}$, Array V, Array W
Seiteneffekte : Datenstruktur, welche die Partition verwaltet

```
1  n ← |M|
2  Für v ∈ M
3  |  MAKESET(v)
4  l ← 1
5  solange l ≤ n^2 und D[l] ≤ D_max tue
6  |  v ← V[l]
7  |  w ← W[l]
8  |  p ← FIND(v)
9  |  q ← FIND(w)
10 |  Wenn p ≠ q
11 |  |  UNION(p, q)
12 |  l ← l + 1
```

(b) Es wird maximal n mal MAKESET, $2 \cdot n^2$ mal FIND und n mal UNION aufgerufen, jeweils angewendet auf eine Menge von maximal n Elementen. Die Laufzeit der FIND–Operationen dominiert die Laufzeit der MAKESET– und UNION–Operationen sowie die der restlichen Schritte des Algorithmus, die in $O(n^2)$ laufen. Als Folge von $2n^2$ Operationen in UNION–FIND ergibt sich somit eine Laufzeit von $O(n^2 \cdot G(n))$.

Problem 3: Alternative zu Heaps 2pt

Beschreibung der Datenstruktur. Eine Datenstruktur, die die gegeben Anforderungen erfüllt, ist ein Array der Länge k, dessen Einträge jeweils auf eine doppelt verkettete Liste zeigen. Siehe dazu auch Abbildung 1.

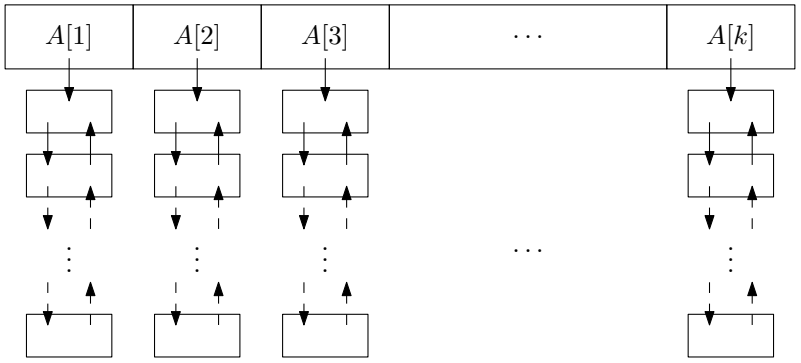

Abbildung 1: Datenstruktur Array of Doubly Linked Lists

Für jede mögliche Prioritätszahl $i \in \{1, \ldots, k\}$ gibt es in dem Array genau ein Feld, das wir den Elementen mit der Priorität i zuordnen. Die zu i gehörige Liste enthält also nur Elemente der Priorität i.

Da jedes Element in der doppelt verketteten Liste konstant viel Speicher verbraucht, und wir zusätzlich nur $k \in O(n)$ weitere Werte in dem Array vorhalten müssen, ist der Platzbedarf der Datenstruktur in $O(n)$.

Beschreibung der Operationen. Um ein Element mit Priorität i einzufügen, brauchen wir nur auf die i–te Liste zugreifen, und das Element an den Anfang der Liste hängen. Beide Operationen sind in $O(1)$ möglich.

Das Löschen eines Elementes ist in einer doppelt verketteten Liste ebenfalls in $O(1)$ möglich. Da jedes Element seinen Nachfolger und Vorgänger kennt, muss nur die Verzeigerung so umgebogen werden, dass der Vorgänger auf den Nachfolger zeigt – und umgekehrt. Dies erfordert eine konstante Anzahl an Operationen pro Element.

Um das Element höchster Priorität zu finden iterieren wir sukzessive von hinten beginnend durch das Array und prüfen für jeden Eintrag, ob die zugehörige Liste mindestens ein Element enthält. Da eine nicht leere Liste nur Elemente gleicher Priorität enthält, können wir, sobald wir eine nicht leere Liste gefunden haben, einfach das erste Element aus der Liste auswählen und zurückgeben. Sei c der konstante Aufwand um das erste Element einer Liste zurückzugeben. So benötigen wir im Worst–case, nämlich falls keine Elemente in der Datenstruktur sind, den Aufwand $k \cdot c \in O(k)$ um die Suche durchzuführen.

Folgende Algorithmen in Pseudocode realisieren die Verwaltung der Datenstruktur. Dabei sei $A[1, \ldots, k]$ das Array, dass die doppelt verketteten Listen verwaltet. Die Operation ADD fügt ein neues Element an den Kopf einer Liste, während FIRST das Kopfelement einer Liste zurückgibt.

Algorithmus 2 : INSERT(i, v)

Eingabe : Zu speichender Wert v mit Priorität i
Seiteneffekte : Der Wert v befindet sich in der Datenstruktur

```
1 Sei A[1, ..., k] das Array
2 L ← A[i]
3 ADD(L, v)
```

Algorithmus 3 : DELETE(v)

Eingabe : Zu löschender Wert v
Seiteneffekte : v wird aus der Datenstruktur gelöscht

```
1 Nach[Vor[v]] ← Nach[v]
2 Vor[Nach[v]] ← Vor[v]
```

Algorithmus 4 : FINDMAX

Ausgabe : Wert v mit höchster Priorität

```
1 Für i ← k, ..., 1
2     L ← A[i]
3     Wenn L ≠ ∅
4         return FIRST(L)
```

Problem 4: d-Heaps 1pt

(a) Die Speicherung der Werte eines d-Heap erfolgt analog zum 2-Heap: Die Indizes des Arrays entsprechen einer Indizierung der Knoten im Baum von oben (Wurzel) nach unten, in jedem Level des Baumes von links nach rechts. Die Funktionen zur Berechnung der Position des j-ten unmittelbaren Nachfolgers bzw. des Vorgängers des Knoten i, D-SUCC(i, j) und D-PRED(i) ergeben sich folgendermaßen:

$$\text{D-SUCC}(i,j) = d * (i-1) + 1 + j, \quad \text{D-PRED}(i) = \left\lceil \frac{i-1}{d} \right\rceil$$

Für die Herleitung der Indizes von Vorgängern und Nachfolgern in einem d-Heap sei auf die Übung verwiesen.

(b) siehe Abbildung 2, 3.

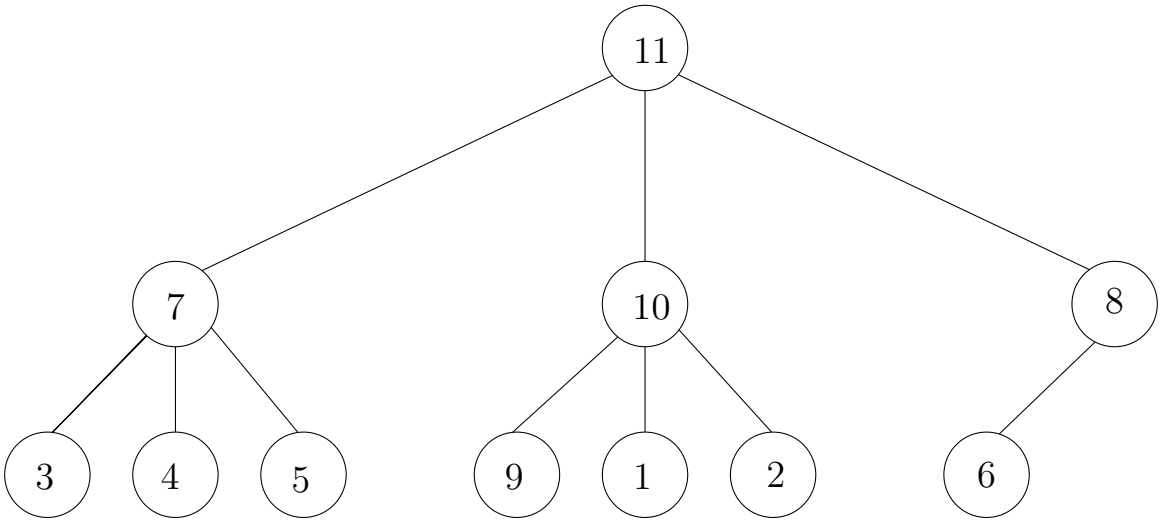

Abbildung 2: Baumdarstellung des 3-Heaps

(c) D-SIFT-UP, siehe Algorihtmus 5, ist analog zu SIFT-UP aus der Vorlesung definiert, wobei lediglich die Berechnung des Vorgängerknoten geändert werden muss. Der worst-case tritt ein, wenn sich der Wert eines Blattes ändert und entlang des Pfades vom Blatt zur Wurzel getauscht werden muss. Die Anzahl der Tauschoperationen ergibt sich damit aus der Höhe des Baumes zu $O(log_d n)$.

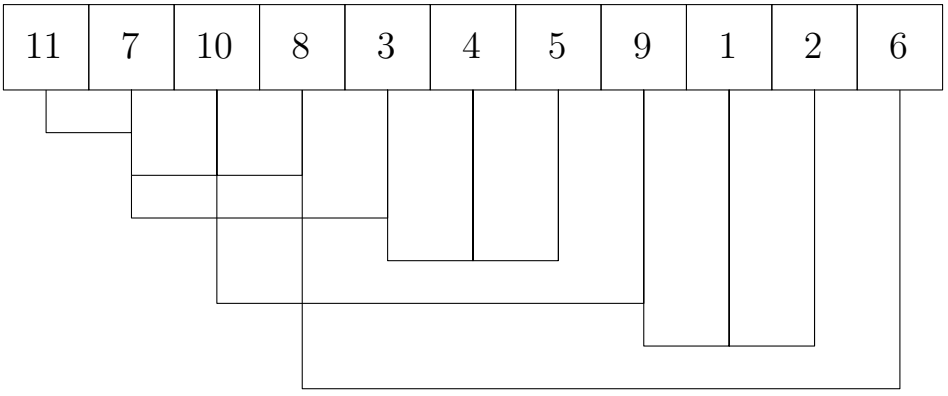

Abbildung 3: Arraydarstellung des 3-Heaps

Algorithmus 5 : D-SIFT-UP(A, i)

Eingabe : Baum als Array A, D-HEAP-Eigenschaft erfüllt, bis auf evtl. in i
Ausgabe : Das Array A als D-HEAP

```
1 ℓ ← i
2 solange D-PRED(ℓ) > 0 und A[ℓ] > A[D-PRED(ℓ)] tue
3 |   Vertausche A[ℓ] und A[D-PRED(ℓ)]
4 |   ℓ ← D-PRED(ℓ)
```