Institut für Theoretische Informatik
Lehrstuhl Prof. Dr. D. Wagner

Lösungsvorschlag 1

Vorlesung Algorithmentechnik im WS 09/10

Problem 1: Dynamisches Array (Amortisierte Analyse) [vgl. Kapitel 0.3 im Skript]

Ein dynamisches Array wird zunächst als leeres Array A der Länge $l(A) = 0$ initialisiert. Falls INSERT (x) ein bereits volles Array A vorfindet, wird ein neues Array A' mit doppelter Länge angelegt und die bereits vorhandenen Elemente werden aus dem vorherigen Array in das neue Array kopiert. Schließlich wird auch das eigentlich einzufügende Element x nach A' kopiert. Das Array A' hat dann also die Länge $l(A') = 2\ l(A)$ und enthält $l(A) + 1$ Elemente. Falls nach DELETE (x) das Array A nur noch zu maximal einem Viertel gefüllt ist, so wird ein neues Array A' mit halber Länge angelegt und die übrigen Elemente aus dem vorherigen Array werden in das neue Array kopiert. Das Array A' hat dann also die Länge $l(A') = \lfloor \frac{1}{2}\ l(A) \rfloor$ und enthält $\lfloor \frac{1}{4}\ l(A) \rfloor$ Elemente (Hinweis: Da die Arraylängen Zweierpotenzen sind, ist jede Länge $l(A) \geq 4$ durch 4 teilbar). In allen anderen Fällen wird x einfach mit konstanten Kosten eingefügt bzw. gelöscht. Gelöschte Elemente geben Platz frei, der durch Einfügungen wieder gefüllt wird, bevor es zu einer Verdoppelung der Arraylänge kommt. Eine entsprechende Speicherplatzverwaltung wird als gegeben vorausgesetzt.

Analysieren Sie die amortisierten Kosten einer Folge von INSERT (x) und DELETE (x). Setzen Sie für das Kopieren eines Elements konstante Kosten voraus.

Lösung. Analyse mit der Potentialmethode

Allgemeine Vorgehensweise:

- Analysiere Dynamik der gegebenen Struktur ...
- Definiere Potentialfunktion $\mathcal{C} : D_i \mapsto \mathcal{C}(D_i)$ mit $\mathcal{C}(D_n) \geq \mathcal{C}(D_0)$ für beliebiges n. Dabei sei D_i bzw. A_i die Datenstruktur bzw. das Array nach der i-ten Operation.
- Zeige, dass die amortisierten Kosten eine **obere Schranke** der tatsächlichen Kosten sind (also $\mathcal{C}(D_n) \geq \mathcal{C}(D_0)$).
- Zeige für jeden Operatiostyp, dass die amortisierte Kosten $\hat{c}_i$ **asymptodisch beschränkt** (z.B. konstant) sind (falls i-te Operation vom jeweiligen Typ ist).

Analyse der Dynamik:
Das Array sei zu Beginn leer, d.h. Inhalt $\#(A_0) = 0$, Länge $l(A_0) = 0$ und Füllgrad $\alpha(A_0) = 1$. Die erste INSERT-Operation setzt $l(A_1) := 1$ und fügt das erste Element ein. Eine DELETE-Operation bei Länge $l(A_{i-1}) = 2$ und Inhalt $\#(A_{i-1}) = 1$ löscht das eine vorhandene Element und setzt $l(A_i) := 0$ (statt $l(A_i) = \frac{1}{2}\, l(A_{i-1}) = 1$). Ist das Array exakt zur Hälfte gefüllt, so befindet es sich in einer Art Gleichgewicht. Vor einer Verdopplung müssen zuerst nochmaleinmal soviele Elemente eingefügt werden, wie schon enthalten sind. Diese können die Kopierkosten der schon beinhalteten Elemente mittragen (mit konstantem Zusatzaufwand pro Element). Umgekehrt müssen vor einer Verkürzung zuerst die Hälfte der enthaltenen Elemente gelöscht werden. Diese können ebenfalls die Kopierkosten der noch übrigen Elemente mittragen (mit konstantem Zusatzaufwand pro Element).

Daher sei das Potential im Gleichgewichtszustand gleich Null. Mit jedem eingefügten Element soll das Potential um mindestens 2 wachsen, damit die Kosten des Umkopierens doppelt so vieler Elemente (wie im Gleichgewichtszustand enthalten) gedeckt sind. Mit jedem gelöschten Element soll das Potential um midestens 1 wachsen, damit die Kosten des Umkopierens von halbsovielen Elementen (wie im Gleichgewichtszustand) gedeckt sind.

Definition der Potentialfunktion:

$$\mathcal{C}(D_i) = \begin{cases} \frac{1}{2}\, l(A_i) - \#(A_i), & \text{für } \alpha < \frac{1}{2} \\ 2\,\#(A_i) - l(A_i), & \text{für } \alpha \geq \frac{1}{2} \end{cases}$$

- Gleichgewichtszustand: $\alpha = \frac{1}{2} \rightsquigarrow$

$$\begin{aligned} \mathcal{C}(D_{i-1}) &= 2\,\#(A_{i-1}) - l(A_{i-1}) \\ &= 2\,\frac{1}{2}\, l(A_{i-1}) - l(A_{i-1}) = 0 \end{aligned}$$

- Insert bei Füllgrad $\alpha \geq \frac{1}{2}$ aber ohne Umkopieren:
  Inhalt des Arrays wächst um 1, Länge des Arrays bleibt gleich $\rightsquigarrow$

$$\begin{aligned} \mathcal{C}(D_i) &= 2(\#(A_{i-1}) + 1) - l(A_i) \\ &= 2\,\#(A_{i-1}) - l(A_{i-1}) + 2 = \mathcal{C}(D_{i-1}) + 2 \end{aligned} \tag{1}$$

  Das Potential wächst also um 2.

- Delete bei Füllgrad $\alpha < \frac{1}{2}$ aber ohne Umkopieren:
  Inhalt des Arrays schrumpft um 1, Länge des Arrays bleibt gleich $\rightsquigarrow$

$$\begin{aligned} \mathcal{C}(D_i) &= \frac{1}{2}\, l(A_i) - (\#(A_{i-1}) - 1) \\ &= \frac{1}{2}\, l(A_{i-1}) - \#(A_{i-1}) + 1 = \mathcal{C}(D_{i-1}) + 1 \end{aligned} \tag{2}$$

  Das Potential wächst also um 1.

- **vor** Insert mit Umkopieren: $\alpha = 1 \rightsquigarrow$

$$\begin{aligned} \mathcal{C}(D_{i-1}) &= 2\,\#(A_{i-1}) - l(A_{i-1}) \\ &= 2\, l(A_{i-1}) - l(A_{i-1}) = l(A_{i-1}) \end{aligned} \tag{3}$$

- **vor** Delete mit Umkopieren:
  Inhalt des Arrays ist $\lfloor \frac{1}{4}\, l(A_{i-1}) \rfloor + 1$.
  Für $l(A_{i-1}) > 4$ ergibt dies einen Füllgrad $\alpha < \frac{1}{2}$ und die Rundung wird hinfällig $\rightsquigarrow$

$$\begin{aligned} \mathcal{C}(D_{i-1}) &= \frac{1}{2}\, l(A_{i-1}) - \#(A_{i-1}) \\ &= \frac{1}{2}\, l(A_{i-1}) - (\frac{1}{4}\, l(A_{i-1}) + 1) = \frac{1}{4}\, l(A_{i-1}) - 1 \end{aligned} \tag{4}$$

Obere Schranke:

- D_0 ist Zustand vor erster Operation: Inhalt ist Null und Länge ist Null, also $\mathcal{C}(D_0) = 0$
- D_n ist Zustand nach n-ter Operation: $\mathcal{C}(D_n) \geq 0$, da Potentialfunktion nicht negativ ist.

Damit $\mathcal{C}(D_n) \geq \mathcal{C}(D_0) \rightsquigarrow$ Amortisierte Kosten sind obere Schranke für tatsächliche Kosten.

Beschränktheit amortisierter Kosten:

- INSERT ohne Umkopieren:

  $\alpha(A_{i-1}) \geq \frac{1}{2},\ \alpha(A_i) \geq \frac{1}{2}$, tritt nur auf für $l(A_{i-1}) > 2$:

  $$\begin{aligned} \hat{c}_i &= c_i + (\mathcal{C}(D_i) - \mathcal{C}(D_{i-1})) \overset{(1)}{=} c_i + 2 = 1 + 2 \\ &= 3 \end{aligned}$$

  $\alpha(A_{i-1}) < \frac{1}{2},\ \alpha(A_i) < \frac{1}{2}$, tritt nur auf für $l(A_{i-1}) > 8$:

  $$\begin{aligned} \hat{c}_i &= c_i + \mathcal{C}(D_i) - \mathcal{C}(D_{i-1}) \\ &= 1 + (\frac{1}{2}\, l(A_i) - \#(A_i)) - (\frac{1}{2}\, l(A_{i-1}) - \#(A_{i-1})) \\ &= 1 + \frac{1}{2}\, l(A_{i-1}) - \#(A_{i-1}) - 1 - \frac{1}{2}\, l(A_{i-1}) + \#(A_{i-1}) \\ &= 0 \end{aligned}$$

  $\alpha(A_{i-1}) < \frac{1}{2},\ \alpha(A_i) = \frac{1}{2}$, tritt nur auf für $l(A_{i-1}) > 4$:

  $$\begin{aligned} \hat{c}_i &= c_i + \mathcal{C}(D_i) - \mathcal{C}(D_{i-1}) \\ &= 1 + (2\,\#(A_i) - l(A_i)) - (\frac{1}{2}\, l(A_{i-1}) - \#(A_{i-1})) \\ &= 1 + 2\,\frac{1}{2}\, l(A_i) - l(A_i) - \frac{1}{2}\, l(A_i) + \frac{1}{2}\, l(A_i) - 1 \\ &= 0 \end{aligned}$$

- DELETE ohne Umkopieren:

  $\alpha(A_{i-1}) < \frac{1}{2}, \alpha(A_i) \geq \frac{1}{2}$, tritt nur auf für $l(A_{i-1}) > 8$:

  $$\begin{aligned} \hat{c}_i &= c_i + (\mathcal{C}(D_i) - \mathcal{C}(D_{i-1})) \overset{(2)}{=} c_i + 1 = 1 + 1 \\ &= 2 \end{aligned}$$

  $\alpha(A_{i-1}) \geq \frac{1}{2},\ \alpha(A_i) \geq \frac{1}{2}$, tritt nur auf für $l(A_{i-1}) > 1$:

  $$\begin{aligned} \hat{c}_i &= c_i + \mathcal{C}(D_i) - \mathcal{C}(D_{i-1}) \\ &= 1 + (2\,\#(A_i) - l(A_i)) - (2\,\#(A_{i-1}) - l(A_{i-1})) \\ &= 1 + 2\,\#(A_{i-1}) - 2 - l(A_{i-1}) - 2\,\#(A_{i-1}) + l(A_{i-1}) \\ &= -1 \end{aligned}$$

  $\alpha(A_{i-1}) = \frac{1}{2},\ \alpha(A_i) < \frac{1}{2}$, tritt nur auf für $l(A_i) > 4$:

  $$\begin{aligned} \hat{c}_i &= c_i + \mathcal{C}(D_i) - \mathcal{C}(D_{i-1}) \\ &= 1 + (\frac{1}{2}\, l(A_i) - \#(A_i)) - (2\,\#(A_{i-1}) - l(A_{i-1})) \\ &= 1 + \frac{1}{2}\, l(A_{i-1}) - \frac{1}{2}\, l(A_{i-1}) + 1 - 2\,\frac{1}{2}\, l(A_{i-1}) + l(A_{i-1}) \\ &= 2 \end{aligned}$$

- INSERT mit Umkopieren:

  $\alpha(A_{i-1}) = 1,\ \alpha(A_i) \geq \frac{1}{2}$, tritt auf für $l(A_{i-1}) \geq 0$:

$$\begin{aligned}
\hat{c}_i &= c_i + \mathcal{C}(D_i) - \mathcal{C}(D_{i-1}) \overset{(3)}{=} c_i + \mathcal{C}(D_i) - l(A_{i-1}) \\
&= c_i + (2\,\#(A_i) - l(A_i)) - l(A_{i-1}) \\
&= c_i + 2\,l(A_{i-1}) + 2 - 2\,l(A_{i-1}) - l(A_{i-1}) \\
&= l(A_{i-1}) + 1 + 2 - l(A_{i-1}) \\
&= 3
\end{aligned}$$

- DELETE mit Umkopieren:

  $\alpha(A_{i-1}) < \frac{1}{2},\ \alpha(A_i) = \frac{1}{2}$, tritt nur auf für $l(A_{i-1}) > 4$:

$$\begin{aligned}
\hat{c}_i &= c_i + \mathcal{C}(D_i) - \mathcal{C}(D_{i-1}) \overset{(4)}{=} c_i + \mathcal{C}(D_i) - \frac{1}{4}\,l(A_{i-1}) + 1 \\
&= c_i + (2\,\#(A_i) - l(A_i)) - \frac{1}{4}\,l(A_{i-1}) + 1 \\
&= c_i + 2\,\frac{1}{4}\,l(A_{i-1}) - \frac{1}{2}\,l(A_{i-1}) - \frac{1}{4}\,l(A_{i-1}) + 1 \\
&= \frac{1}{4}\,l(A_{i-1}) + 1 - \frac{1}{4}\,l(A_{i-1}) + 1 \\
&= 2
\end{aligned}$$

  Alle weiteren Fälle für $l(A_{i-1}) \leq 4$ ergeben ohnehin konstante amortisierte Kosten, da alle Summanden ≤ 8 sind.

$\Rightarrow$ Insgesamt ergibt sich ein amortisierter Aufwand in $O(1)$.

Lösung. Analyse mit der Buchungsmethode

Idee: Wenn das Array mit Länge $l(A_{i-1})$ verdoppelt wird, so müssen zuvor mindestens $l(A_{i-1})/2$ Elemente eingefügt worden sein, die noch nie Umkopiert wurden. Diese Einfügungen können die Kosten des Umkopierens nach der Verdopplung mittragen. Wenn das Array mit Länge $l(A_{i-1})$ halbiert wird, so müssen zuvor mindestens $l(A_{i-1})/4$ Elemente gelöscht worden sein. Diese Löschungen können die Kosten des Umkopierens nach der Halbierung mittragen.

Definiere die Kosten wie folgt:

	tatsächliche Kosten	**amortisierte Kosten**
INSERT(x)	1 Einfügen von x	3 Einfügen von x + Umkopieren von x + Umkopieren eines schon vorhandenen Elements in A
DELETE(x)	1 Löschen von x	2 Löschen von x + Umkopieren eines noch vorhandenen Elements in A
INSERT MIT UMKOPIEREN	$l(A_{i-1}) + 1$	3
DELETE MIT UMKOPIEREN	$\lfloor l(A_{i-1})/4 \rfloor + 1$	2

Durch die Veranschlagung höherer amortisierter Kosten im Vergleich zu den tatsächlichen Kosten von INSERT wird vor einer Verdopplung ein Kredit von $\geq (3-1)(l(A_{i-1})/2) = l(A_{i-1})$ angehäuft.

Damit reduzieren sich die tatsächlichen für INSERT MIT UMKOPIEREN zu $l(A_{i-1})+1-l(A_{i-1})=1$. Amortisiert erhöht sich dies zu 3, da INSERT MIT UMKOPIEREN dieselben amortisierten Kosten wie INSERT haben soll.
Durch die Veranschlagung höherer amortisierter Kosten im Vergleich zu den tatsächlichen Kosten von DELETE wird vor einer Halbierung ein Kredit von $\geq (2-1)(l(A_{i-1})/4) = l(A_{i-1})/4$ angehäuft. Damit reduzieren sich die tatsächlichen für DELETE MIT UMKOPIEREN zu $\lfloor l(A_{i-1})\rfloor + 1 - l(A_{i-1})/4 \leq 1$. Amortisiert erhöht sich dies zu 2, da DELETE MIT UMKOPIEREN dieselben amortisierten Kosten wie DELETE haben soll.

$\Rightarrow$ Insgesamt ergibt sich ein amortisierter Aufwand in $O(1)$.

Problem 2: Master-Theorem (Laufzeit rekursiver Funktionen) [vgl. Kapitel 0.4.5 im Skript]

Prüfen Sie die Anwendbarkeit der beiden Master-Theorem-Versionen (Master-Theorem und seine allgemeinere Form) auf die folgenden Rekurrenzgleichungen *(auch Rekursionsgleichungen genannt)*:

(a) $T(n) = 2\,T(\lfloor n/2 \rfloor) + 2\,T(\lceil n/2 \rceil) + n^2$

(b) $T(n) = 7\,T(n/7) + n\,\ln(n)$

Begründen Sie Ihre Antwort.

Lösung. zu (a)
Umformung der Rekurrenzgleichung zu $4\,T(n/2) + n^2$.

- **Allgemeinere Form des Master-Theorems:**

$$\begin{aligned} f(n) &= n^2 \in \Theta(n^k) \quad \text{mit } k = 2 \geq 0 \\ 0 < \alpha_i &= \frac{1}{2} < 1 \quad \text{für } i = 1, \ldots, 4 \\ \sum_{i=1}^{4} (\alpha_i)^k &= 4\left(\frac{1}{2}\right)^2 = 4\,\frac{1}{4} = 1 \end{aligned}$$

  $\Rightarrow$ Fall (ii) anwendbar: $T(n) \in \Theta(n^2 log(n))$

- **Master-Theorem:**

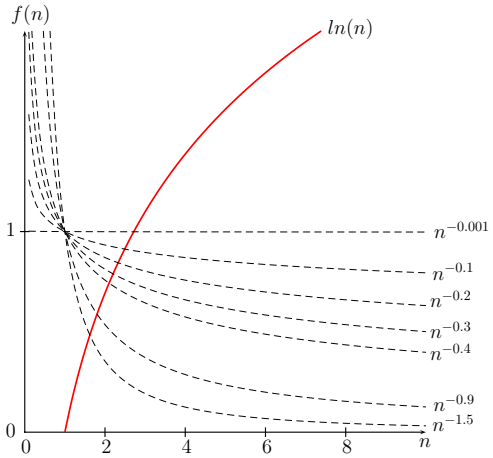
$$\begin{aligned} a &= 4 \geq 1 \\ b &= 2 > 1 \\ log_2(4) &= 2 \\ f(n) &= n^2 \in \Theta(n^2) = \Theta(n^{log_2(4)}) \end{aligned}$$

  $\Rightarrow$ Fall (ii) anwendbar: $T(n) \in \Theta(n^2 log(n))$

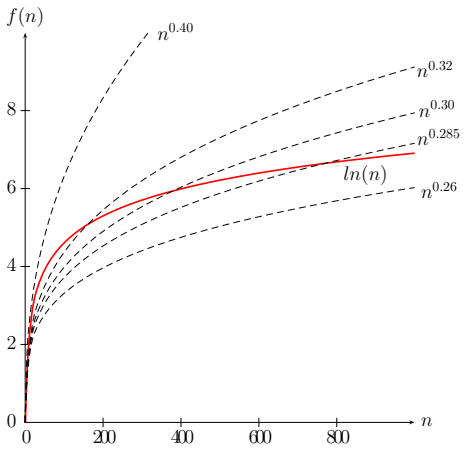
Lösung. zu (b)

- **Master-Theorem:**

$$\begin{aligned} a &= 7 \geq 1 \\ b &= 7 > 1 \\ log_7(7) &= 1 \\ f(n) &= n\ln(n) \notin \Theta(n) = \Theta(n^{log_7(7)}), \quad \text{Fall (ii)} \\ f(n) &= n\ln(n) \notin O(n^{1-\epsilon}) = O(n^{log_7(7)-\epsilon}) \quad \forall \epsilon > 0, \quad \text{da } \ln(n) \notin O(1/n^{\epsilon}), \\ & \quad \text{Fall (iii), (vgl. Abbildung)} \\ f(n) &= n\ln(n) \notin \Omega(n^{1+\epsilon}) = \Omega(n^{log_7(7)+\epsilon}) \quad \forall \epsilon > 0, \quad \text{da } \ln(n) \notin \Omega(n^{\epsilon}), \\ & \quad \text{Fall (i), (vgl. Abbildung)} \end{aligned}$$



(a) zu Fall (iii).



(b) zu Fall (i).

⇒ Master-Theorem **nicht** anwendbar!

- **Allgemeinere Form des Master-Theorems:**

$$f(n) = n\ln(n) \notin \Theta(n^k) \quad \forall k \geq 0, \quad \text{da } \ln(n) \notin \Omega(n^{\epsilon})$$

⇒ Allgemeinere Form des Master-Theorems **nicht** anwendbar!

Problem 3: Binäre Suche (Laufzeit rekursiver Funktionen) [vgl. Kapitel 0.4.5 im Skript]

Gegeben sei eine sortierte Sequenz $A := (a_1, \ldots, a_n)$ von n Elementen. Die Binäre Suche sucht nun nach einem Element x in A, indem sie x mit dem mittleren Element in A vergleicht und so entscheidet, ob und wenn ja in welcher Hälfte die Suche rekursiv fortgesetzt wird.

Modellieren Sie die Binäre Suche in Pseudocode und erstellen Sie die zugehörige Rekurrenzgleichung. Bestimmen Sie eine asymptotisch scharfe obere und eine asymptotisch scharfe untere Schranke der Laufzeit.

Lösung. Die Sortierungsrichtung der Sequenz ist in der Aufgabenstellung nicht vorgegeben. Der hier vorgestellte Pseudocode setzt eine aufsteigende Sortierung der Werte voraus.

Algorithmus 1 : BinärSuche($A[1 \dots n], x$)

```
1  Ausgabe Sequenzindex von x bzw. −1 falls x nicht enthalten ist
2  Wenn n = 1 und x ≠ A[1]
3  |  return −1                                              // Abbruchkriterium
4  sonst
5  |  Wenn x = A[⌊n/2⌋]
6  |  |  return ⌊n/2⌋                                       // Abbruchkriterium
7  |  sonst
8  |  |  Wenn x < A[⌊n/2⌋]
   |  |  |                                                  // Aufsteigende Sortierung
9  |  |  |  BinärSuche(A[1 ... ⌊n/2⌋ − 1], x)              // Rekursionsaufruf
10 |  |  sonst
11 |  |  |  BinärSuche(A[⌊n/2⌋ + 1 ... n], x)              // Rekursionsaufruf
```

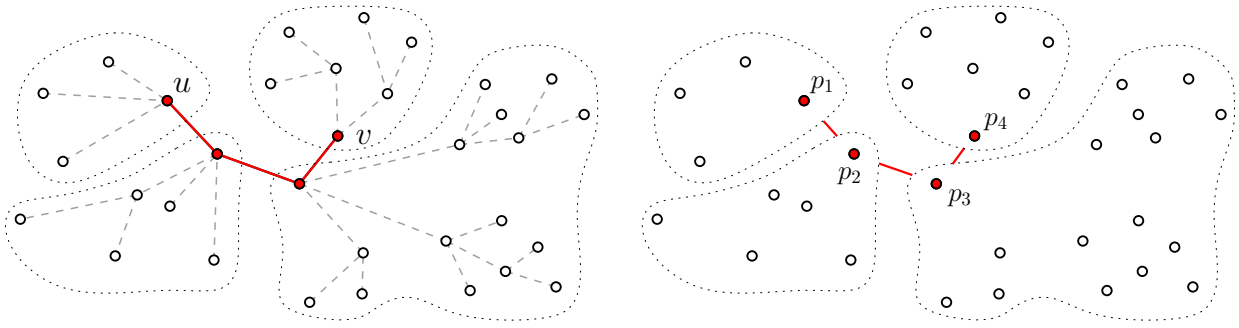
- **Rekurrenzgleichung:** $T(n) = T(n/2) + f(n)$ mit $f(n)$ konstant
- **Allgemeinere From des Mastertheorems:**

$$\begin{aligned} f(n) &\in \Theta(1) = \Theta(n^k) \quad \text{mit } k = 0 \geq 0 \\ 0 < \alpha &= \frac{1}{2} < 1 \\ (\alpha_i)^k &= 1 \end{aligned}$$

$\Rightarrow$ Fall (ii) anwendbar: $T(n) \in \Theta(n^0 log(n)) = \Theta(log(n))$
Damit ist $\Omega(log(n))$ ist eine asymptotisch scharfe untere und $O(log(n))$ eine asymptotisch scharfe obere Schranke.

Problem 4: Pfad aus Superknoten (Union-Find) [vgl. Kapitel 1.1 und 1.2 im Skript]

Gegeben sei ein Baum T mit n Knoten und ein Knotenpaar $\{u, v\}$. Schreiben Sie einen Algorithmus in Pseudocode, der den (eindeutigen) Pfad zwischen u und v in T berechnet und für jeden Pfadknoten p_i die Teilbäume mit Wurzel p_i kontrahiert (siehe untenstehende Abbildung). Benutzen Sie dazu eine modifizierte Tiefensuche. Speichern Sie besuchte Knoten in einer Union-Find-Struktur, in der Knotenmengen vereinigt werden, die zu Teilbäumen des selben Pfadknotens gehören.



(a) Pfadknoten mit Teilbäumen

(b) Knotenmengen nach Kontraktion der Teilbäume.

Lösung. **Idee:** Verwende eine rekursive, modifizierte Tiefensuche. Starte bei Knoten u. Durchsuche alle Unterbäume des aktuell betrachteten Knotens danach, ob sie v enthalten. Speichere alle besuchten Knoten in der Pfadliste. Falls v im Unterbaum nicht gefunden wird, lösche die Knoten des Unterbaums wieder aus der Pfadliste und kontrahiere den Unterbaum, d.h. vereinige die Knotenmenge des Unterbaums mit dem aktuell betrachteten Knoten (Die Union-Find-Struktur speichert zu Beginn jeden Knoten des Baumes als einelementige Menge). Die Tiefensuche sucht den kompletten Baum ab (auch wenn v bereits gefunden ist), denn die Unterbäume von v müssen ebenfalls kontrahiert werden.

Bemerkung:
Pfad-Liste ist gobal.
Union-Find-Struktur ist global, speichert zu Beginn jeden Knoten des Baumes als einelementige Menge.

Algorithmus 2 : getPathAndSupernodes(u, v, T)

```
1  Seiteneffekt Pfad als Liste, Superknoten als Union-Find
2  markiere u als besucht
3  hänge u an Pfad an
4  Wenn u = v
5  |  Vereinige alle Knoten - fertig!
6  sonst
7  |  DFS-path(u, v, T)                          // durchsuche UBs von u
```

Rekursiver Teil des Pseudocodes

Algorithmus 3 : DFS-path(r, v, T)

```
1  Ausgabe gefunden als Boolsche Variable
2  gefunden ← false                              // vor Abstieg in UBs von r
3  solange r noch unmarkierten Nachbarn s hat tue
4  |  inUnterbaumGefunden ← false                // vor Abst. in UBs von s
5  |  markiere s
6  |  hänge s an Pfad an
7  |  Wenn s = v
8  |  |  gefunden ← true                        // weitersuchen für Kontaktion
9  |  inUnterbaumGefunden = DFS-path(s, v, T)   // Abstieg von s
10 |  Wenn inUnterbaumGefunden
11 |  |  gefunden ← true                        // UBs von s sind UBs von r
12 |  Wenn nicht inUnterbaumGefunden und nicht gefunden
13 |  |  Lösche letztes Element aus Pfad        // Aufstieg von s nach r
14 |  |  Union(Find(r), Find(s))
15 return gefunden
```

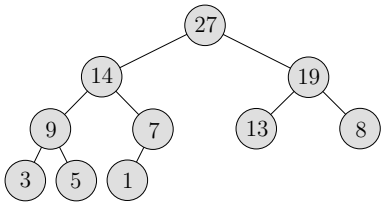
Problem 5: Heap-Varianten (Heap-Datenstruktur) [vgl. Kapitel 1.3 im Skript]

(a) Die klassische Heap-Eigenschaft $A[\mathit{Vorgänger}[i]] \geq A[i]$ kann auch als Max-Heap-Eigenschaft bezeichnet werden. Der klassische Heap wird somit zu einem Max-Heap. Wie sieht konsequenterweise die Heap-Eigenschaft eines Min-Heaps aus?

(b) Sei A ein Max-Heap (gespeichert in einem Array) und A' enthalte die Elemente aus A in umgekehrter Reihenfolge. Ist A' ein Min-Heap? Begründen Sie Ihre Antwort.

(c) Ergänzen Sie an untenstehendem Heap A (Max-Heap in Baumform) die zugehörigen Array-

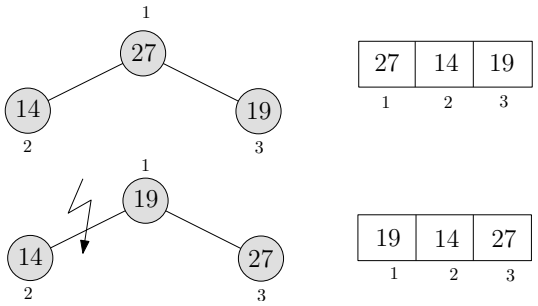
Indizes. Wenden Sie DELETE $(A, 2)$ an, indem Sie die einzelnen Zustände, die der Baum durchläuft, darstellen.



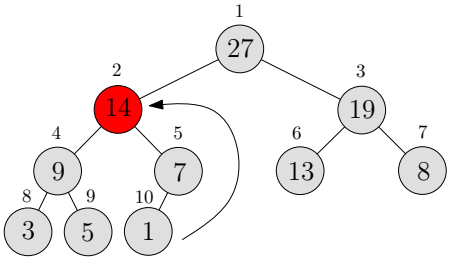
Lösung.

(a) **Min-Heap-Eigenschaft:** $A[\mathit{Vorgänger}[i]] \leq A[i]$

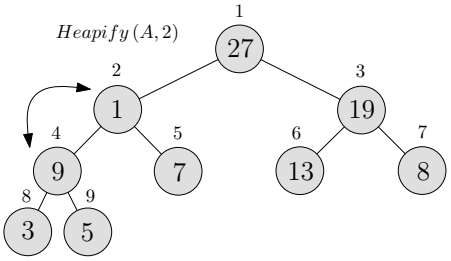
(b) Min-Heap durch Umkehrung? **Antwort:** Nein! (siehe Gegenbeispiel)



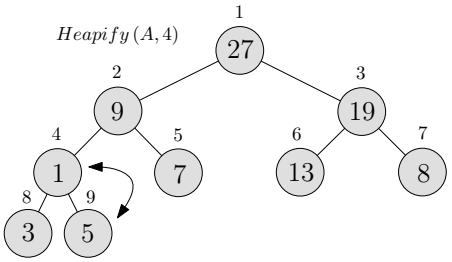
(c) **Array-Indizes und Anwendung von Delete $(A, 2)$:**
Beachte: Indizierung beginnt mit 1 (nicht mit 0). DELETE (A, i) hat als Parameter i den Index (nicht den Wert) des Knotens.



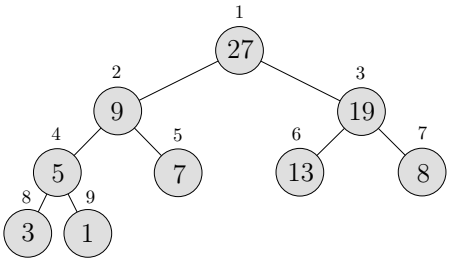
(a) Löschung des Knotens mit Index 2.



(b) HEAPIFY$(A, 2)$.



(a) HEAPIFY$(A, 4)$.



(b) Fertig!