FernUniversität in Hagen

Lösungsvorschläge zur Hauptklausur „1661 Datenstrukturen I“

7.8.2010

Aufgabe 1

(a)

algebra *circle*

sorts *bool, real, circle, point*

ops

createPoint:	*real* × *real*	→	*point*
distancePoint:	*point* × *point*	→	*real*
createCircle:	*point* × *real*	→	*circle*
center:	*circle*	→	*point*
radius:	*circle*	→	*real*
containsPoint:	*circle* × *point*	→	*bool*
containsCircle:	*circle* × *circle*	→	*bool*
intersectsCircle:	*circle* × *circle*	→	*bool*
equalCircles:	*circle* × *circle*	→	*bool*
extendCircle:	*circle* × *circle*	→	*circle*
minimumCircle:	*point* × *point*	→	*circle*
distanceCircle:	*circle* × *circle*	→	*real*

(b)

sets $point = \{(x, y) \mid x, y \in real\} = real \times real = \mathbb{R}^2$

$circle = \{(p, r) \mid p \in point, r \in real, 0 \leq r\} = point \times \mathbb{R}_0^+$

(c)

Sei im Folgenden:

$p = (x, y)$ und $p_i = (x_i, y_i)$, für $1 \leq i \leq 2$

$p_1 = p_2 \Leftrightarrow x_1 = x_2 \wedge y_1 = y_2$

und

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

functions

createPoint(x, y) $= (x, y)$

distancePoint(p_1, p_2) $= d(p_1, p_2)$

createCircle(p, r) $= (p, \max(0, r))$

center(p, r) $= p$

radius(p, r) $= r$

containsPoint$((p_1, r_1), p)$ $= d(p_1, p) \leq r_1$

containsCircle$((p_1, r_1), (p_2, r_2))$ $= d(p_1, p_2) + r_2 \leq r_1$

intersectsCircle$((p_1, r_1), (p_2, r_2))$ $= (d(p_1, p_2) \leq r_1 + r_2)$

equalCircles$((p_1, r_1), (p_2, r_2))$ $= (p_1 = p_2 \wedge r_1 = r_2)$

extendCircle$((p_1, r_1), (p_2, r_2))$

$$= \begin{cases} (p_1, r_1) & \text{, falls } d(p_1, p_2) + r_2 \leq r_1 \\ (p_1, d(p_1, p_2) + r_2) & \text{, sonst.} \end{cases}$$

minimumCircle(p_1, p_2) $$= \left(\left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2} \right), \frac{d(p_1, p_2)}{2} \right)$$

distanceCircle$((p_1, r_1), (p_2, r_2))$

$$= \begin{cases} 0 & \text{, falls } d(p_1, p_2) < r_1 + r_2 \\ d(p_1, p_2) - r_1 - r_2 & \text{, sonst.} \end{cases}$$

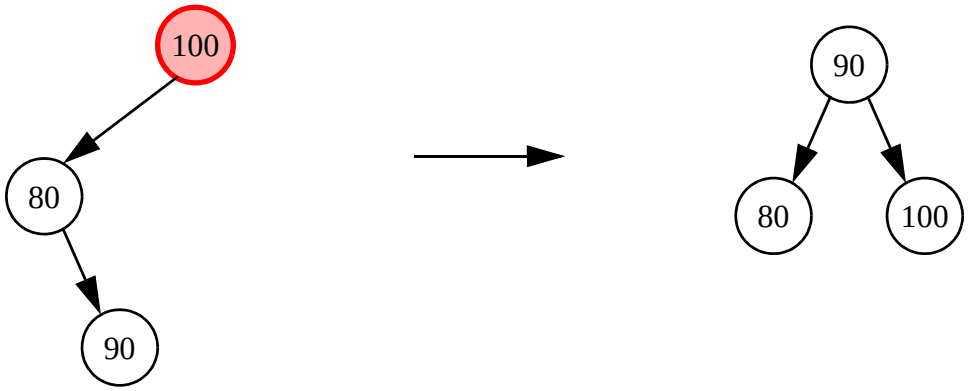
end *circle*

Anmerkung: Alternative Lösungen sind natürlich zulässig.

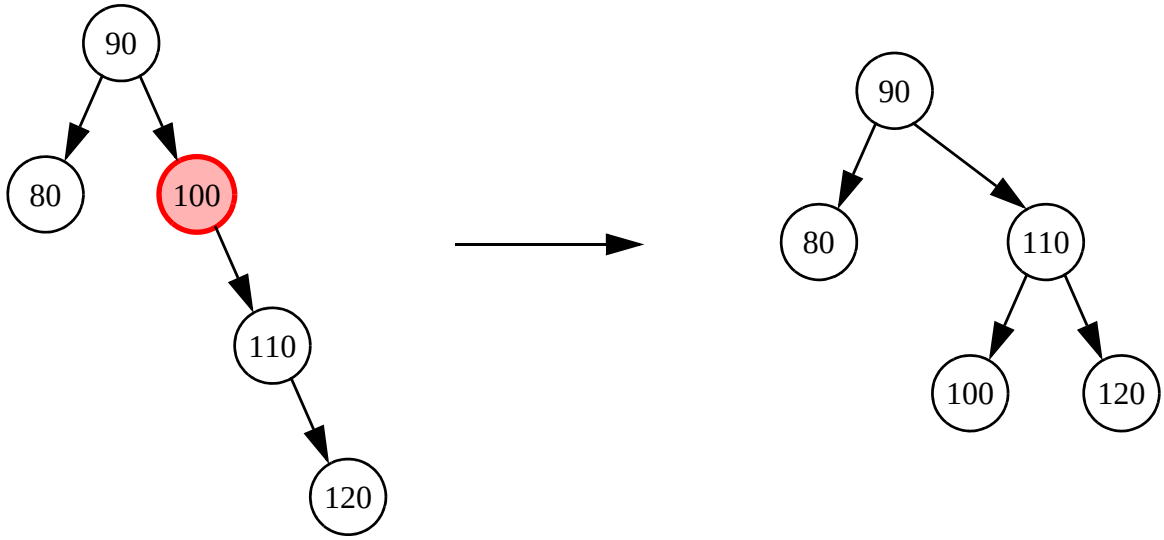
Aufgabe 2

(a)

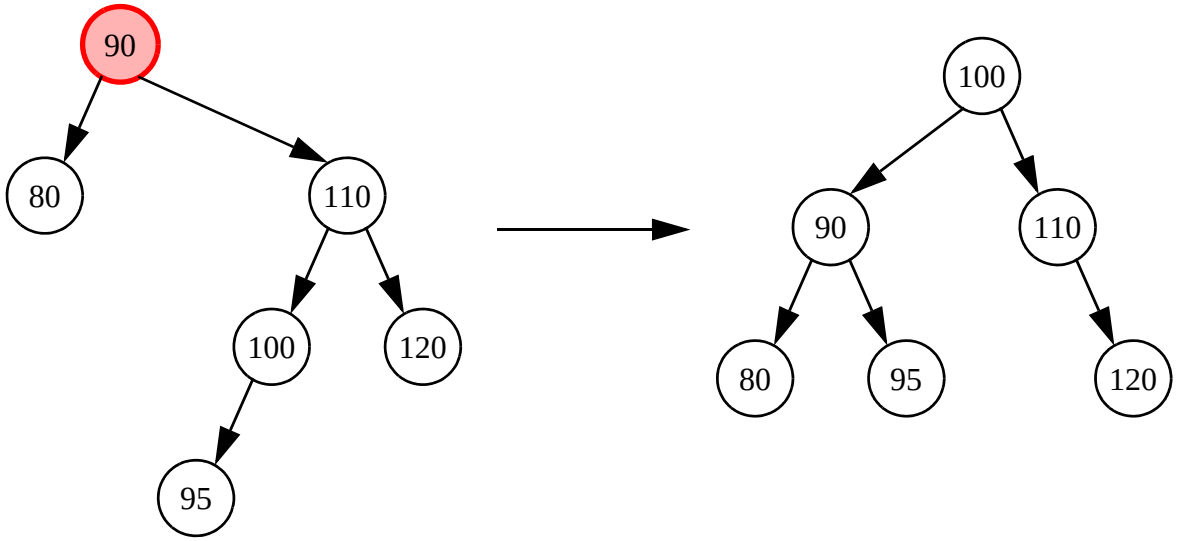
Die erste Rebalancierung erfolgt beim Einfügen des Elements 90 in den Baum. Es ist eine Doppelrotation durchzuführen:



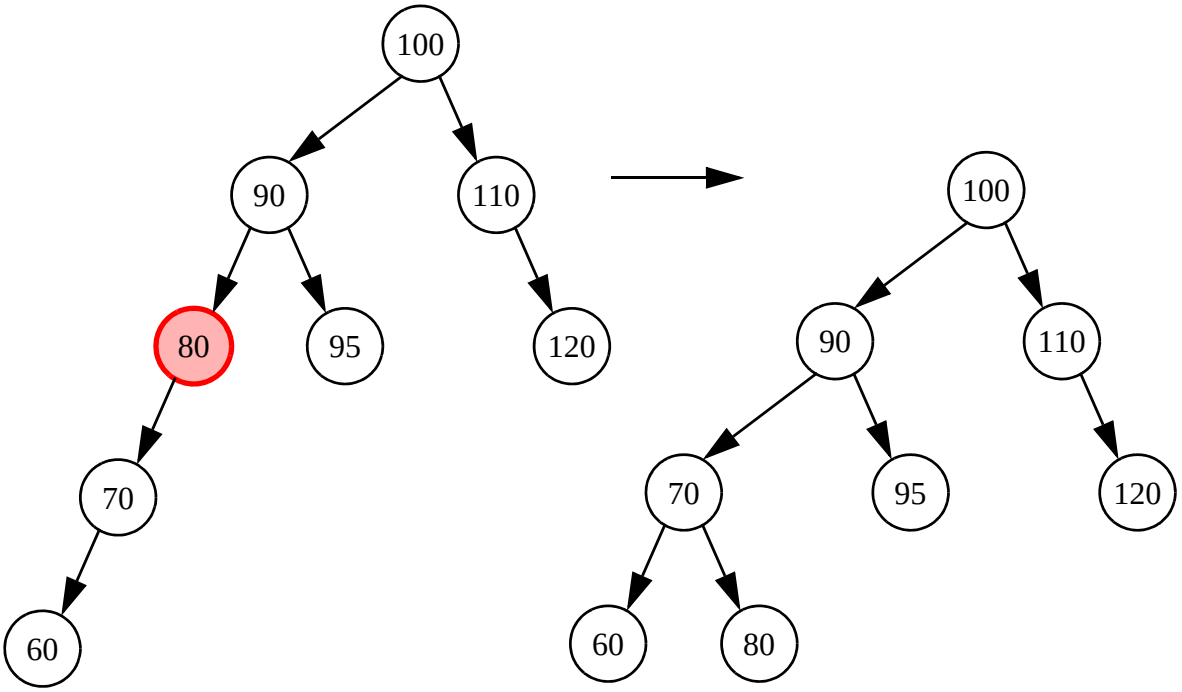
Das Element 110 lässt sich problemlos einfügen. Die Balanceverletzung, die beim Einfügen des Elements 120 auftritt, wird mittels einer einfachen Rotation ausgeglichen:



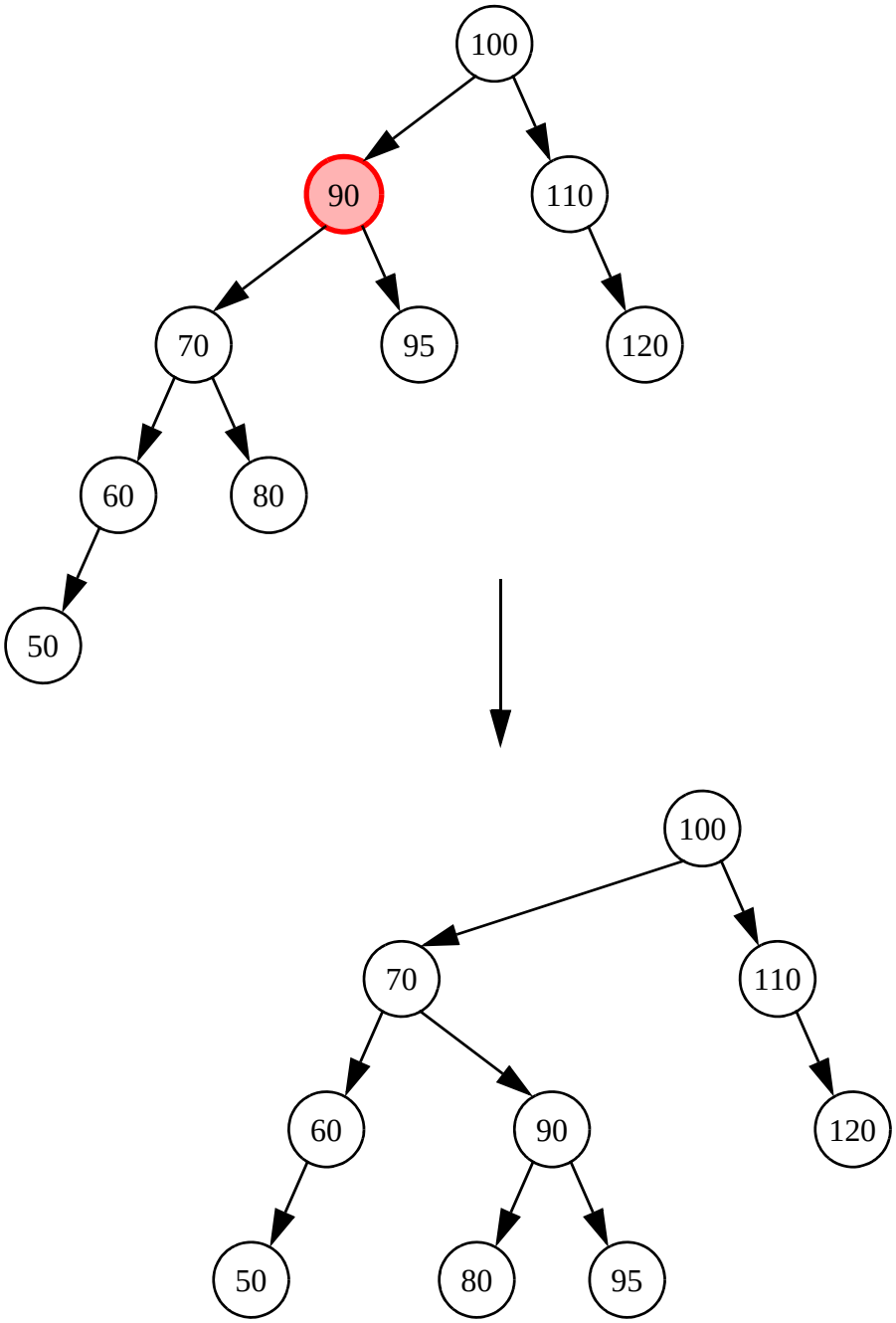
Beim Einfügen des Elements 95 kommt es erneut zur Verletzung der Balance am Wurzelknoten. Diese wird durch eine Doppelrotation korrigiert:



Beim Einfügen des Elements 60 muss wieder eine einfache Rotation durchgeführt werden, um den Baum auszugleichen:

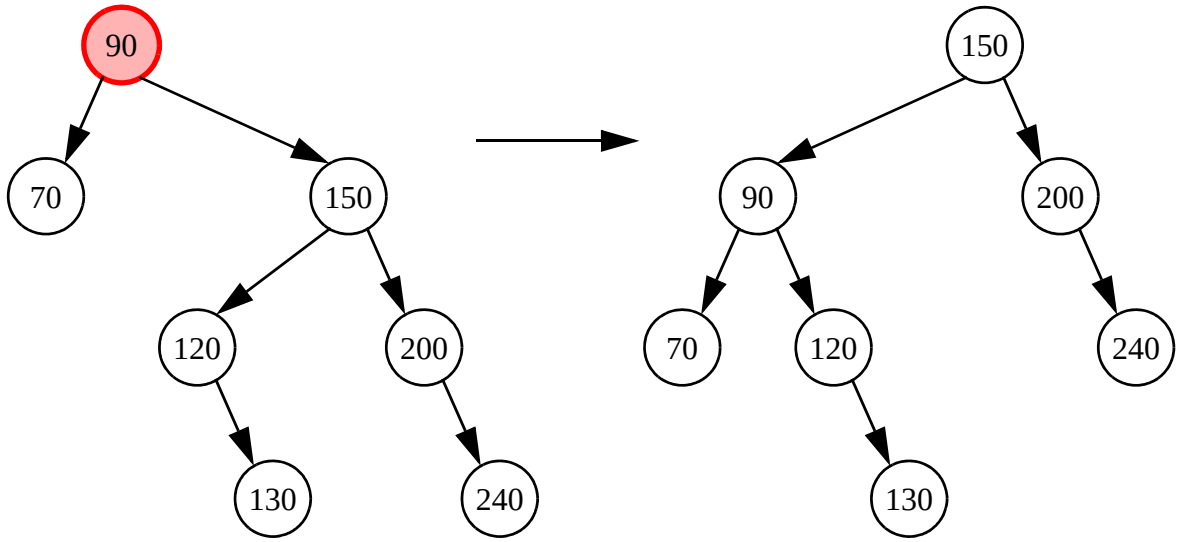


Auch beim Einfügen des Elements 50 wird eine einfache Rotation benötigt:

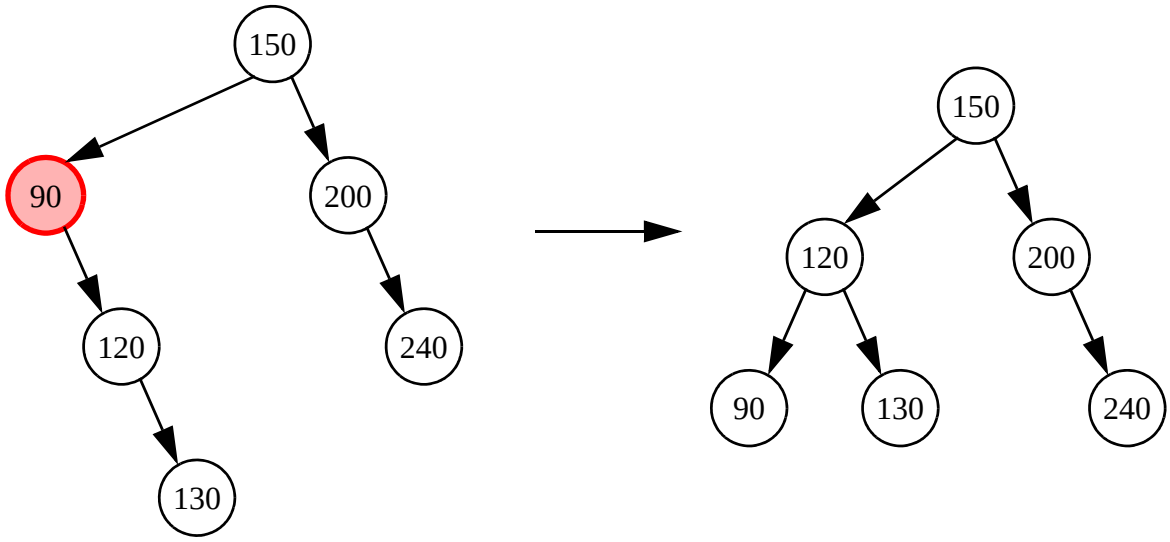


(b)

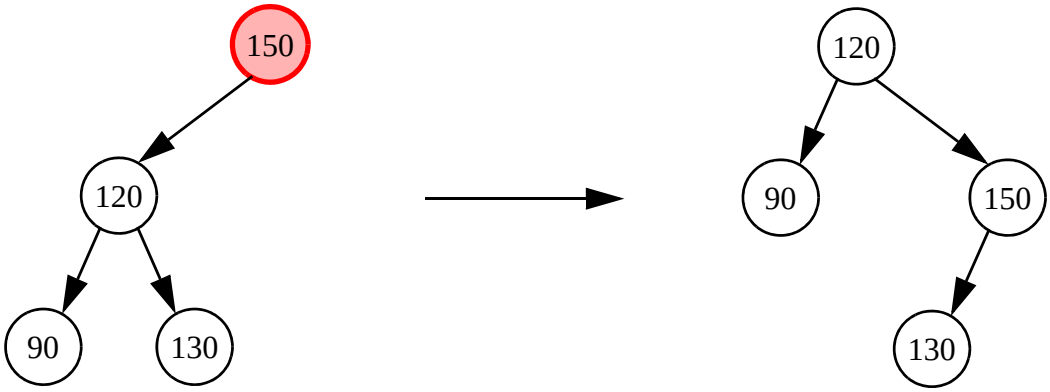
Beim Löschen des mit 80 markierten Knotens gerät die Wurzel aus der Balance. Es wird eine einfache Rotation benötigt, um diesen Zustand zu beheben:



Beim Löschen von 70 wird ebenfalls eine einfache Rotation benötigt:



Das Löschen des Elements 240 benötigt keine Rebalancierungen. Beim Löschen von 200 gerät der Wurzelknoten aus der Balance, was durch eine einfache Rotation ausgeglichen wird:



(c)

Wir gehen davon aus, dass man auf die Bestandteile eines nicht-leeren Knotens eines AVL-Baums mittels *left*, *right* und *elem* zugreifen kann.

Damit lässt sich der Algorithmus wie folgt formulieren:

```
algorithm printdesc( t )
begin
   if isempty(t) then return; end if
   printdesc(t.right);
   output(t.elem);
   printdesc(t.left);
end printdesc.
```

(d)

Der geforderte Algorithmus kann wie folgt lauten:

```
algorithm printrange(t, min, max)
begin
   if isempty(t) then return; end if
   if t.elem > min then
      printrange(t.left, min, max);
   end if
   if min ≤ t.elem and t.elem ≤ max then
      output(t.elem);
   end if
   if t.elem < max then
      printrange(t.right, min, max);
   end if
end printrange;
```

Der Algorithmus sucht im Baum zunächst das kleinste Element, welches größer oder gleich zu *min* ist. Diese Suche benötigt O(log(*n*)). Anschließend findet eine Art inorder-Durchlauf statt, der nach Erreichen der *max*-Schwelle abgebrochen wird. Dabei wird jeder Knoten der Ergebnismenge einmal erreicht. Dieser Vorgang benötigt demnach eine Laufzeit von O(*e*). Daraus ergibt sich die Gesamtlaufzeit von O(log(*n*)+*e*).

Aufgabe 3

(a)

Die initiale Verteilung ergibt:

$B_{Ø}$={Karl, Egon, Katrin, Hein, Boris, Anton, Karla, Diana, Peter, Nora}

B_{a}={Barbara}

Der nächste Durchlauf liefert folgende Behälterinhalte:

$B_{Ø}$={Karl, Egon, Hein, Boris, Anton, Karla, Diana, Peter, Nora}

B_{n}={Katrin}

B_{r}={Barbara}

Danach haben die Behälter folgende Inhalte:

$B_{Ø}$={Karl, Egon, Hein, Nora}

B_{a}={Karla, Diana, Barbara}

B_{i}={Katrin}

B_{n}={Anton}

B_{r}={Peter}

B_{s}={Boris}

Die nächste Phase ergibt die folgenden Behälterinhalte:
B_{a}={Nora}
B_{b}={Barbara}
B_{e}={Peter}
B_{i}={Boris}
B_{l}={Karl, Karla}
B_{n}={Egon, Hein, Diana}
B_{o}={Anton}
B_{r}={Katrin}

Die Neuverteilung liefert:
B_{a}={Diana}
B_{i}={Hein}
B_{o}={Egon}
B_{r}={Nora, Barbara, Boris, Karl, Karla}
B_{t}={Peter, Anton, Katrin}

Der erneute Durchlauf liefert:
B_{a}={Barbara, Karl, Karla, Katrin}
B_{e}={Hein, Peter}
B_{g}={Egon}
B_{i}={Diana}
B_{n}={Anton}
B_{o}={Nora, Boris}

Bei der letzten Verteilung erhält man:
B_{a}={Anton}
B_{b}={Barbara, Boris}
B_{d}={Diana}
B_{e}={Egon}
B_{h}={Hein}
B_{k}={Karl, Karla, Katrin}

B_n={Nora}
B_p={Peter}

Hieraus ergibt sich die sortierte Folge:
Anton Barbara Boris Diana Egon Hein Karl Karla Katrin Nora Peter

(b)

Da es soviele Durchläufe gibt, wie der längste String lang ist, benötigt man l_{max} Durchläufe.

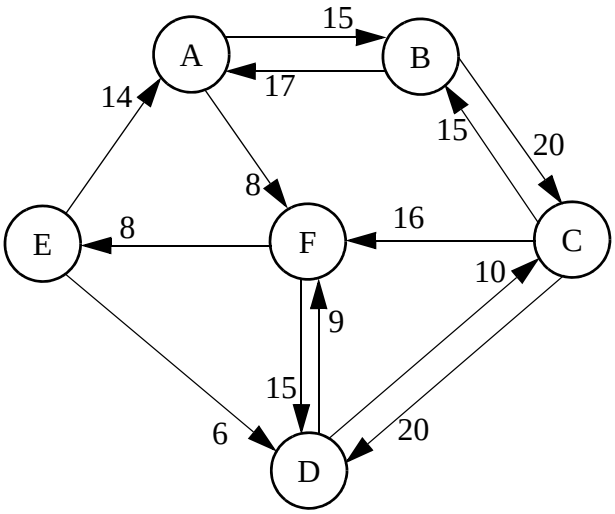
(c)

Um Strings nicht jedes Mal zu prüfen und in den Behälter $B_{Ø}$ zu verschieben, kann man zuächst alle Strings gemäß ihrer Länge in Behälter speichern, d.h. es gibt einen Behälter für Strings der Länge 1, einen Behälter für Strings der Länge 2 usw. Diese Behälter werden in einer Priority Queue verwaltet, wobei die jeweilige Stringlänge die Priorität angibt. Während der Sortierung mittels Radixsort, werden die String im obersten Behälter der Queue mit zur Verarbeitung verwendet, wenn die dazugehörige Länge der aktuell verwendeten Zeichenposition entspricht.

Aufgabe 4 Eisenbahn-Magnat

(a)

Es ergibt sich folgender gerichteter, kantengewichteter Graph:



(b)

Wir verwenden den Algorithmus von Dijkstra, um nacheinander für jeden Knoten (=Station) X im Graph den kürzesten Weg zu jedem Knoten Y zu finden. Die Kosten k des jeweils kürzesten Weges von X nach Y werden als Tupel/Zeilen (X, Y, k) in die Ergebnistabelle eingetragen. Bei n Knoten v_1, …, v_n wird Dijkstras Algorithmus also nacheinander für jeden Knoten v_i einmal aufgerufen.

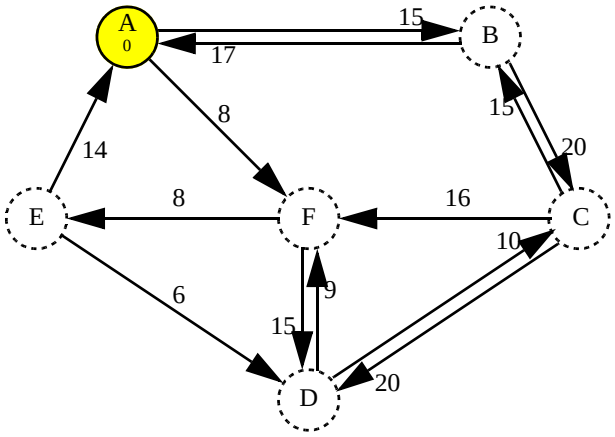
(c)

Bei n Knoten muss Dijkstras Algorithmus n-mal aufgerufen werden. Aus dem Kurs wissen wir, dass bei einer Implementierung mit Adjazenzmatrix der Aufwand $O(n^2)$, bei einer Implementierung mit Adjazenzlisten $O(e \log n)$ ist. Wir wissen jedoch weiter, dass der Ausgangsgrad jedes Knotens im Graphen maximal 5 ist. Daher gilt $e \leq 5n$ und die einzelne Laufzeit ist $O(n \log n)$. Deshalb wählen wir die Implementierung mit Adjazenzlisten und erhalten insgesamt eine Laufzeit von $O(n^2 \log n)$.

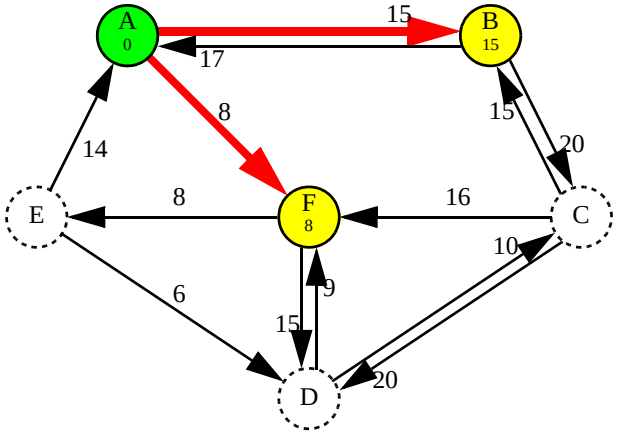
(d)

Wir zeigen nun die Berechnung von Dijkstras Algorithmus auf dem Streckengraphen ausgehend vom Startknoten A. Anstelle der hier gezeigten graphischen Darstellung genügt auch etwa eine tabellarische Darstellung, aus der die einzelnen Veränderungen ersichtlich sind (s.u.).

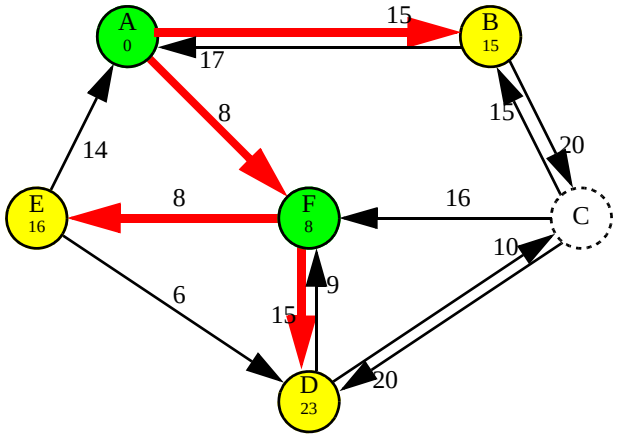
Zuerst wird der Startknoten A gelb und mit der Entfernung 0 markiert:



Nun wird A als nächster Knoten aus der Peripherie grün gefärbt und seine Ausgangskanten verfolgt und rot gefärbt. B und F werden zum ersten Mal erreicht und gelb markiert. Ihre Entfernungen ergeben sich direkt aus den Kantengewichten:

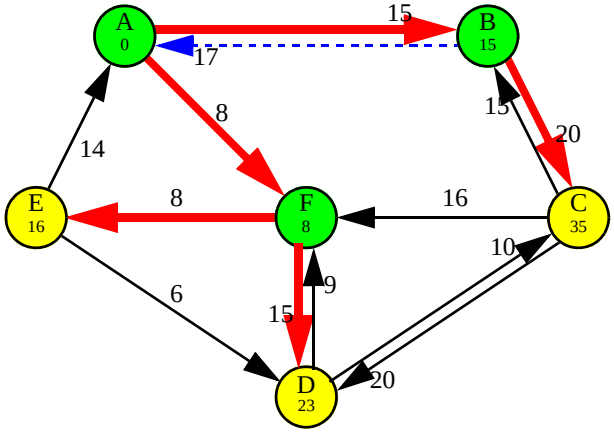


Von allen Knoten in der Peripherie ist nun F der nächste zu A. Er wird grün gefärbt und expandiert. Seine Ausgangskanten führen zu E und D, deren Abstände sich aus dem Abstand von F und den jeweiligen Kantenkosten ergeben. Die neu betrachteten Kanten werden rot:

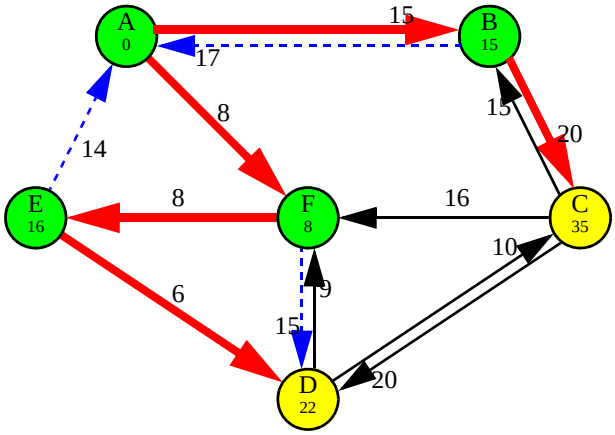


Der nächste gelbe Knoten mit kleinstem Abstand zu A ist B. Er wird grün. Wir färben den neu gefundenen Knoten C gelb, vermerken seinen Abstand von A und färben die Kante (B, C) rot.

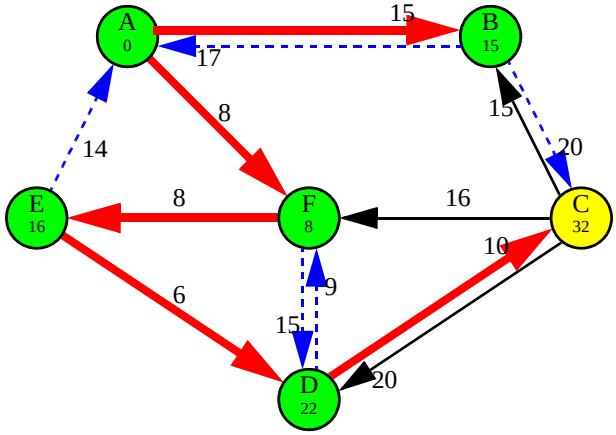
A haben wir bereits grün (d.h. abgearbeitet), daher ist die Kante (B, A) nicht zu untersuchen, wir können sie daher gelb färben (hier blau gestrichelt dargestellt):



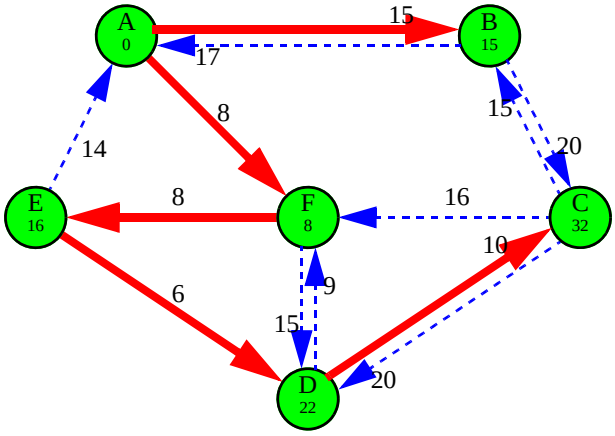
Jetzt ist E der gelbe Knoten minimalen Abstands und wird grün. Wir finden keinen unbesuchten Knoten. Über die nun rot zu färbende Kante (E, D) lässt sich D schneller erreichen als bisher. Daher wird die Kante (F, D) gelb. Auch die Kante (E, A) wird gelb gefärbt, da A bereits grün gefärbt wurde:



Nun wird D grün und expandiert. Kante (D, F) müssen wir nicht berücksichtigen (F ist bereits grün), daher wird sie gelb. Die Kante (D, C) verringert den Abstandswert von C und wird rot. Dafür wird (B, C) gelb gefärbt:



Es verbleibt nur noch Knoten C. Die Kante (C, B) kann gelb werden, ebenso die Kanten (C, F) und (C, D). C wird grün und der Algorithmus terminiert.



Hier noch eine tabellarische Darstellung der Berechnung. Die Abstände für Knoten werden in den Index gesetzt, ungefärbte Knoten und Kanten werden nicht explizit aufgeführt. Neue und veränderte Einträge wurden hervorgehoben.):

Tabelle 1:

Peripherie-knoten (gelb)	abgearbeitete Knoten (grün)	Baumkanten (rot)	ehemalige, ersetzte Baumkanten (gelb)
A_0			
F_8, B_{15}	**A_0**	**(AB)**, **(AF)**	
B_{15}, **E_{16}**, **D_{23}**	A_0, **F_8**	(AB), (AF), **(FE)**, **(FD)**	
E_{16}, D_{23}, **C_{35}**	A_0, F_8, **B_{15}**	(AB), (AF), **(BC)**, (FE), (FD)	**(BA)**
D_{22}, C_{35}	A_0, F_8, B_{15}, **E_{16}**	(AB), (AF), (BC), **(ED)**, (FE)	(BA), **(EA)**, **(FD)**
C_{32}	A_0, F_8, B_{15}, E_{16}, **D_{22}**	(AB), (AF), **(DC)**, (ED), (FE)	(BA), **(BC)**, **(DF)**, (EA), (FD)
	A_0, F_8, B_{15}, E_{16}, D_{22}, **C_{32}**	(AB), (AF), (DC), (ED), (FE)	(BA), (BC), **(CB)**, **(CD)**, **(CF)**, (DF), (EA), (FD)

Es ergibt sich folgender Auschnitt für die Abstandstabelle:

Tabelle 2: Minimale erreichbare Fahrzeiten

Von\Nach	A	B	C	D	E	F
A	0 min	15 min	32 min	22 min	16 min	8 min

Aufgabe 5 Deckblatt

Hier erhalten Sie den Punkt, wenn Sie beide Klausurdeckblätter korrekt und vollständig ausgefüllt haben, also Namen, Matrikelnummer und Adresse korrekt eingetragen und genau diejenigen Aufgaben markiert haben, die Sie auch bearbeitet haben.