Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Dr. Christian Schulz, Dr. Simon Gog
Michael Axtmann

1. Übungsblatt zu Algorithmen II im WS 2016/2017

http://algo2.iti.kit.edu/AlgorithmenII_WS16.php
{christian.schulz,michael.axtmann,sanders,simon.gog}@kit.edu

Musterlösungen

Aufgabe 1 (*Analyse: Kleinaufgaben*)

a) Geben Sie die wesentlichen Unterschiede –laut Vorlesung– zwischen einer (normalen) *Priority Queue* und einer adressierbaren *Priority Queue* an.

b) Vergleichen Sie die Laufzeit einer `merge`-Operation für *Pairing Heaps* und *Binary Heaps*.

Musterlösung:

a) Im Gegensatz zur normalen *Priority Queue* erlaubt die adressierbare *Priority Queue* den direkten Zugriff auf beliebige Elemente über ein *Handle* h. Dieses wird von der `insert`-Operation als Rückgabewert geliefert. Außerdem ermöglicht sie einige zusätzliche Operationen: `remove(h)`, `decreaseKey(h,k)`. Die `merge`-Operation kann prinzipiell auch von normalen *Priority Queues* unterstützt werden, allerdings weniger effizient.

b) In einem *Pairing Heap* wird eine Menge von Bäumen gehalten. Eine `merge`-Operation ist also in konstanter Zeit möglich, indem die jeweiligen Listen vereint werden und der `minPtr` auf das Minimum beider Wälder gesetzt wird. Im *Binary Heap* funktioniert dieses Vorgehen nicht. In der weit verbreiteten Arrayimplementierung gibt es mehrere mögliche Vorgehensweisen. Wenn die Anzahl der Elemente gegeben ist durch n_1 bzw. n_2, $n_1 < n_2$, ergeben sich folgende Laufzeiten:

   (a) Einfügen der kleineren Menge an Elementen: $O(n_1 \cdot \log(n_1 + n_2))$

   (b) Kompletter Neuaufbau: $O(n_1 + n_2)$

   Je nach Verteilung der Elemente können beide Möglichkeiten sinnvoll sein.

Aufgabe 2 (*Analyse: Laufzeitverhalten*)

a) Beweisen Sie allgemein für adressierbare *Priority Queues* die untere Laufzeitschranke von $\Omega(\log n)$ für `deleteMin` unter der Voraussetzung, dass `insert` konstante Laufzeit benötigt.

b) Warum muss diese untere Laufzeitschranke nicht gelten, wenn `insert` mehr Zeit benötigen darf?

Musterlösung:

a) Mit einer Laufzeit von $O(f(n))$ für `deleteMin` ließe sich in Zeit $O(nf(n)) + n \cdot O(1)$ vergleichsbasiert sortieren, indem man zuerst alle Elemente einfügt und dann eines nach dem anderen in aufsteigender Reihenfolge entnimmt. Für `deleteMin` in sublogarithmischer Zeit wäre das ein Widerspruch zur bekannten unteren Schranke für vergleichsbasiertes Sortieren von $\Omega(n \log n)$.

b) `insert` könnte nach jedem Aufruf eine aufsteigend sortierte Liste aller Elemente hinterlassen, mit deren Hilfe sich alle folgenden `min`- und `deleteMin`-Operationen in konstanter Zeit beantworten ließen.

Aufgabe 3 (*Analyse:* best-case *Verhalten*)

a) Geben Sie einen Zustand eines *Fibonacci Heaps* an, für den die nächsten n `deleteMin`-Operationen jeweils konstante Laufzeit benötigen (nicht amortisiert). Begründen Sie Ihre Antwort. Gehen Sie davon aus, dass zwischen den `deleteMin`-Operationen keine anderen Operationen ausgeführt werden.

b) Geben Sie einen Algorithmus an, welcher den von Ihnen angegebene Zustand für beliebige n erzeugt. Beweisen Sie die Korrektheit des Algorithmus.

Musterlösung:

a) Gegeben sei ein *Fibonacci Heap* der Größe n, bestehend aus einem einzelnen Baum. Dieser Baum speichere n Knoten in Form einer verketteten Liste. Die nachfolgenden n `deleteMin`-Operationen führen dann jeweils eine `Cut`-Operation auf den direkten Nachfolgern des jeweiligen Wurzelknotens aus. Da der Wurzelknoten nur einen Nachfolger hat, wird pro `deleteMin`-Operation nur eine `Cut`-Operation ausgeführt. Diese `Cut`-Operationen benötigen jeweils nur konstante Laufzeit. Da der *Fibonacci Heap* vor jeder `deleteMin`-Operation nur aus einem Baum besteht, führt die `deleteMin`-Operation keine `union`-Operation aus. Eine potentiell folgende `union`-Operation auf dem neuen Baum kann durch das Token des alten Wurzelknotens bezahlt werden. Es müssen also auch keine neuen Token für folgende Operationen bezahlt werden.

b) *Behauptung 1*: Die Operation `CreateFibonacciList` erzeugt einen *Fibonacci Heap* F der Größe n, bestehend aus einem einzelnen Baum, repräsentiert durch eine verkettete Liste.

```
1:  function CreateFibonacciList(n ∈ ℕ⁺)
2:      F ← empty fibonacci heap
3:      for i ← n down to 1 do
4:          F.insert(i)
5:          x ← F.insert(i+1)
6:          F.insert(i)
7:          F.deleteMin
8:          F.decreaseKey(x, 0)
9:          F.deleteMin
10:     end for
11: return F
12: end function
```

Beweis: Die Behauptung gilt für $n = 0$, in diesem Fall gibt die Operation `CreateFibonacciList` einen leeren *Fibonacci Heap* zurück.

Invariante 1: Sei $n > 0$ beliebig aber fest. Nach $0 < x \leq n$ Schleifendurchläufen der Zeilen 4-9 besteht F aus einem einzelnen Baum in Form einer verketteten Liste $n - x + 1, \ldots, n$.

Nehmen wir an, dass *Invariante 1* für beliebige aber feste $n > 0$ gilt. Unter dieser Voraussetzung besteht F nach $x = n$ Ausführungen der Zeilen 4-9 aus einem einzelnen Baum in Form einer verketteten Liste $1, \ldots, n$ und *Behauptung 1* gilt somit für $n \in \mathbb{N}_0^+$. Wir beweisen nun *Invariante 1* durch Induktion über die Schleifendurchläufe $x \in \{0, \ldots, n\}$. Sei hierzu $n > 0$ beliebig aber fest.

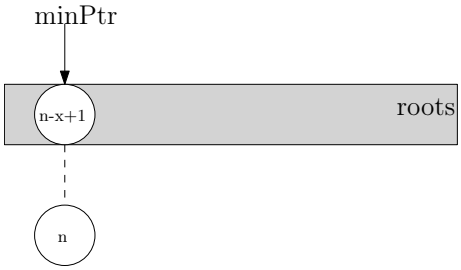
Induktionsanfang: $x = 1$: Nach Ausführung der Zeilen 4-6 enthält F die Elemente "n", "n+1" und "n". Die Zeilen 7-10 entfernen die Elemente "n" und "n+1". Somit enthält F nach dem ersten Schleifendurchlauf das Element "n" und die Invariante gilt für $x = 1$.

Induktionsschritt ($x \to x + 1$): Nach Induktionsvoraussetzung besteht F aus einem Baum, repräsentiert durch eine verkettete Liste, der Form $n - x + 1, \ldots, n$. Die Abbildungen auf der nächsten Seite führen nun die Codezeilen 4-9 aus und zeigen den Zustand von F nach $x + 1$ Iterationen. F besteht nun aus einem Baum, repräsentiert durch eine verkettete Liste, der Form $n - x, \ldots, n$. Somit gilt die Invariante auch nach $x + 1$ Iterationen und der Induktionsschritt $x \to x + 1$ ist abgeschlossen.

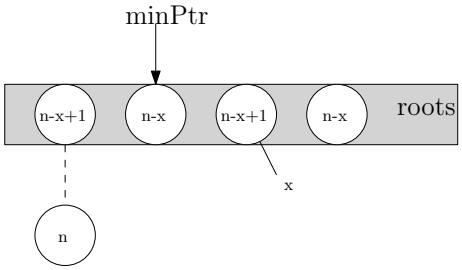
Induktionsschluss: Die Invariante gilt nach jedem Schleifendurchlauf.

Musterlösung:

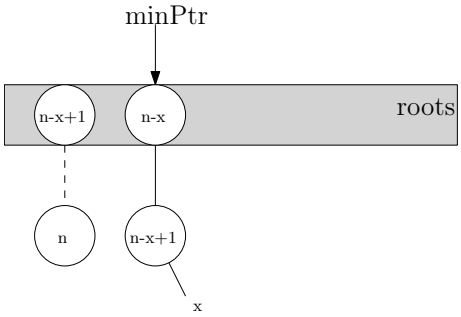
Induktionsvoraussetzung:



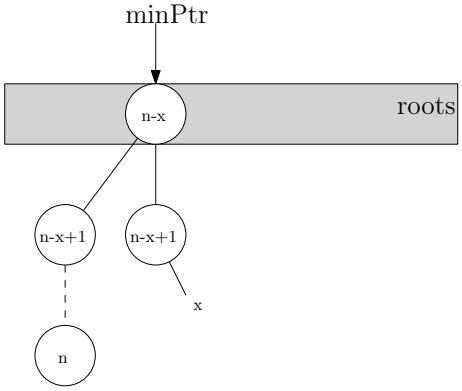
`F.insert(`$n-x$`),x=F.insert(`$n-x+1$`),F.insert(`$n-x$`)`:



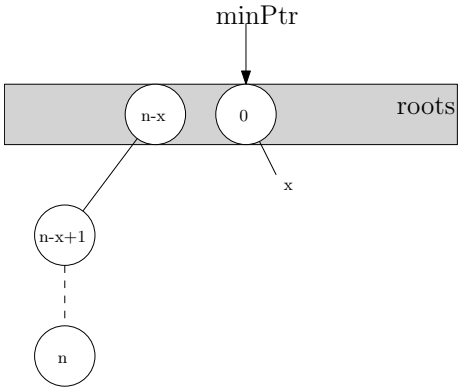
`F.deleteMin` (Zwischenschritt):



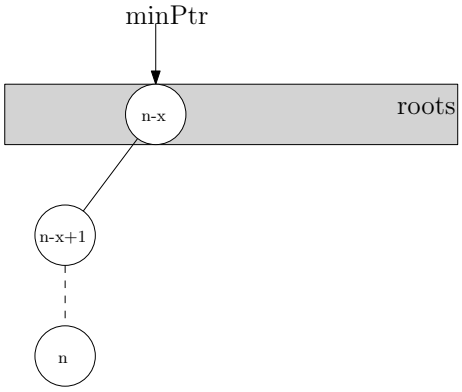
`F.deleteMin`:



`F.decreaseKey(x,0)`:



`F.deleteMin`:



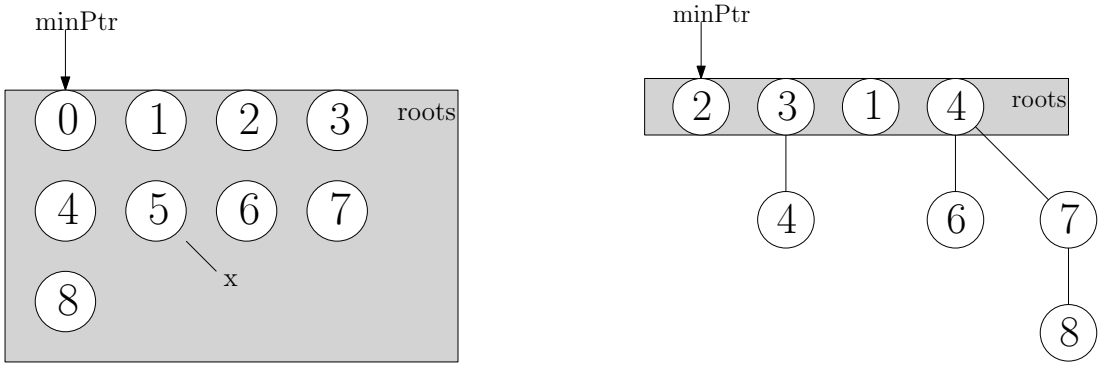
Aufgabe 4 (*Rechnen:* Fibonacci Heaps)

Gegeben sei ein *Fibonacci Heap* mit unten eingezeichnetem Zustand (a).

a) Geben Sie eine möglichst kurze Folge von Operationen an, die diesen Zustand erzeugt.

b) Führen Sie anschließend die Operationen `deleteMin()` auf dem Heap aus. Zeichnen Sie den Zustand des Heaps nach jedem Einfügen eines Baum in ein leeres Bucket und nach jeder Union-Operation.

c) Geben Sie eine möglichst kurze Folge von Operationen an, die den unten eingezeichneten Zustand (b) erzeugt. Tipp: der eingezeichnete Zustand lässt sich aus dem Heap in Abbildung (a) nach der `deleteMin`-Operation durch weitere Operationen erzeugen.



(a) (b)

Musterlösung:

a) Die Folge

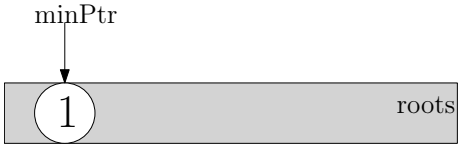
`insert(0)`, `insert(1)`, `insert(2)`, `insert(3)`, `insert(4)`, `insert(5)`, `insert(6)`, `insert(7)`, `insert(8)`

erzeugt den gegebenen Zustand.

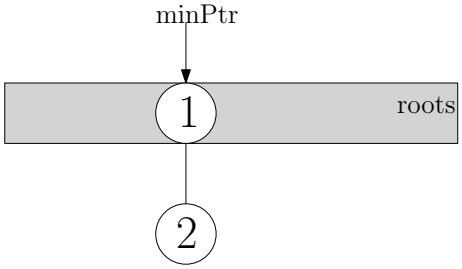
Beachten Sie, dass die Definition des *Fibonacci Heaps* keine Aussage darüber macht, in welcher Reihenfolge Wurzelknoten gespeichert sind. Für diese Lösung und die der nächsten Teilaufgabe nehmen wir eine nach der Reihenfolge der Einfügungeoperationen sortierte Liste von Wurzeln an. Werden Teilbäume abgeschnitten, so werden diese an das Ende der Liste angehängt.

Musterlösung:

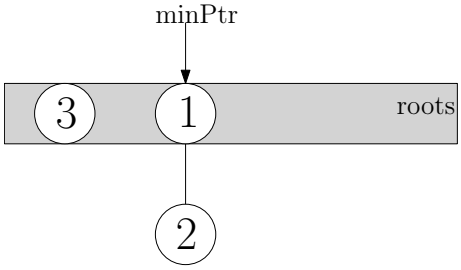
b) h_1 (keine Kollision):



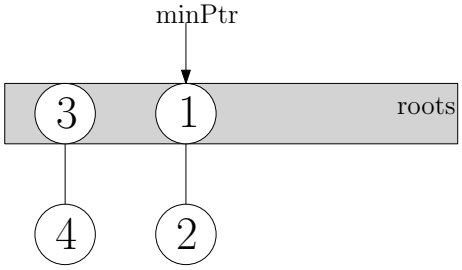
`Union`(h_1, h_2):



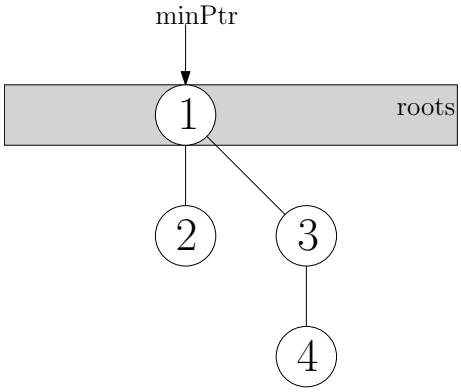
h_3 (keine Kollision):



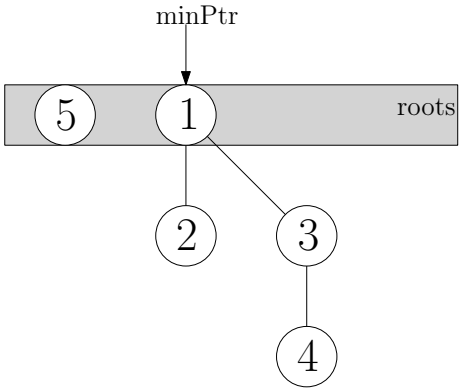
`Union`(h_3, h_4):



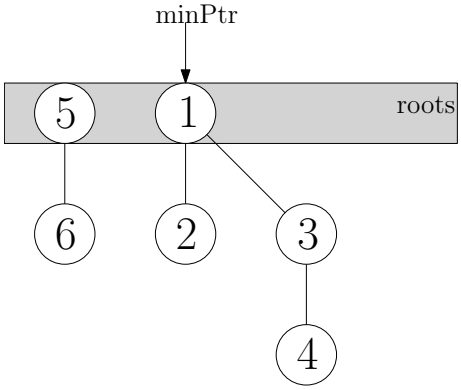
`Union`(h_3, h_1):



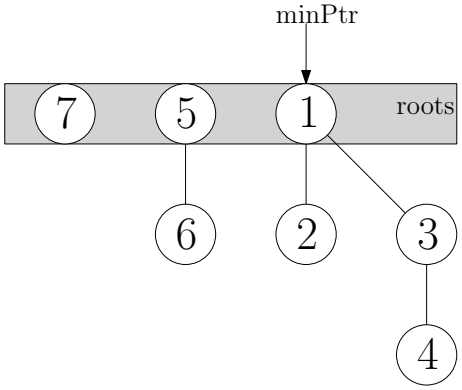
h_5 (keine Kollision):



`Union`(h_5, h_6):

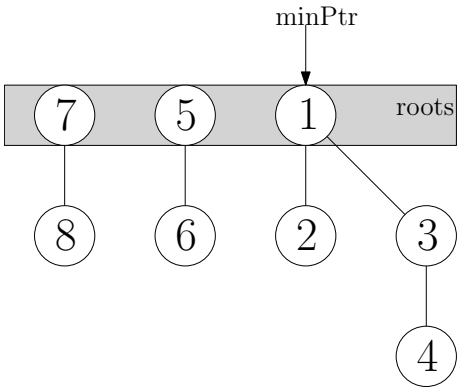


h_7 (keine Kollision):

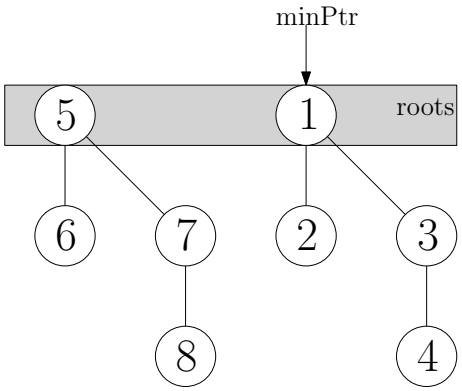


Musterlösung:

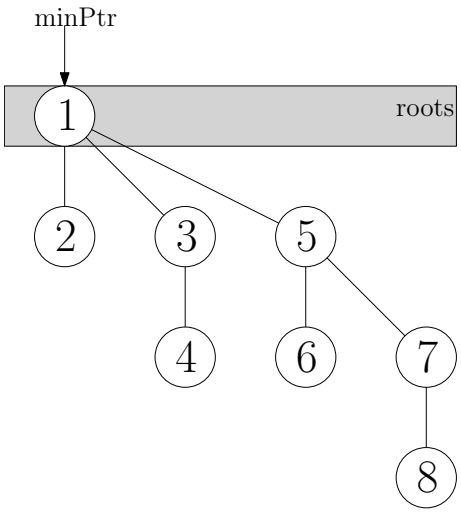
`Union(`h_7, h_8`):`



`Union(`h_7, h_5`):`



`Union(`h_5, h_1`):`



c) Die Folge

`deleteMin()`, `insert(1)`, `decKey(x, 4)`

erzeugt den gegebenen Zustand.

Aufgabe 5 (*Entwurf: Datenstrukturen*)

a) Erweitern Sie die Datenstruktur *Pairing Heap* um die Operation `increaseKey(h: Handle, k: Key)`. Ihre Operation sollte amortisiert $O(\log n)$ Laufzeit benötigen. Geben Sie Pseudocode an. Wie würden Sie bei einem *Binary Heap* vorgehen?

b) Entwerfen Sie eine Datenstruktur welche die Operationen `insert` in $O(\log n)$, Median bestimmen in $O(1)$ und Median entfernen in $O(\log n)$ unterstützt. Eine Beschreibung in Worten ist ausreichend.

Musterlösung:

a) Lösche das Element aus der Datenstruktur ($O(\log n)$) und füge es mit dem geänderten Schlüssel wieder ein ($O(1)$):

```
function IncreaseKey(h : Handle, k : Key)
    remove(h)
    key(h) := k
    insert(h)
end function
```

Bei einem *Binary Heap* würde man zuerst den Schlüssel anpassen und anschließend eine `siftDown`-Operation ausführen.

b) Aufbau des Datentyps:

- Speicherung des aktuellen Median in v.
- Verwendung einer Maximum *Priority Queue* (MaxPQ) für Elemente kleiner als v und einer Minimum *Priority Queue* (MinPQ) für Elemente größer als v.

Bestimmung des Median:
Frage v direkt ab: $O(1)$.

Löschen des Medians:
Ersetze den aktuellen Median v durch das oberste Element der größeren *Priority Queue* (bei Gleichheit verwende MinPQ): `deleteMin` in $O(\log n)$, z.B. mit *Fibonacci Heap*.

Einfügen eines Elements:
Füge das neue Element –in Abhängigkeit von v– in eine der *Priority Queues* ein: `insert` in $O(1)$. Füge v in die kleinere ein (bei Gleichstand verwende MaxPQ): `insert` in $O(1)$. Ersetze v durch das oberste Element der größeren *Priority Queue* (bei Gleichstand verwende MinPQ): `deleteMin` in $O(\log n)$, z.B. mit *Fibonacci Heap*.

Aufgabe 6 (*Entwurf: Anwendung*)

Für ein großes –und wir meinen ein wirklich großes– Fest sind Sie für die Bar zuständig. Für diese Aufgabe haben Sie Ihren eigenen Barroboter entworfen, der automatisch wunderbare Cocktails mischen kann. Die Zutaten dafür werden in großen Kanistern bereitgestellt. Doch genau hier ist das Problem: In all der Hektik des Abends müssen Sie darauf achten, dass kein Kanister leerläuft. Sie wollen den Abend allerdings auch so gut wie möglich genießen und nicht andauernd die Kanister überprüfen. Um dieses Problem zu umgehen, gibt es nur eine Lösung: Eine Nachfüllanzeige muss her! Leider gab es nur noch Anzeigen, die es erlauben eine einzelne Zeile darzustellen.
Es ist klar, dass auf der Anzeige die am dringensten benötigte Zutat angezeigt werden sollte. Ziel ist es also einen Algorithmus zu entwerfen, der die Anzeige immer aktuell hält.

a) Überlegen Sie sich, welche Datenstruktur Sie als Grundlage für Ihren Algorithmus verwenden wollen, um ihn effizient implementieren zu können. Sie können davon ausgehen, dass die für einen Cocktail benötigten unterschiedlichen Zutaten wesentlich weniger sind als die Gesamtmenge an vorhandenen unterschiedlichen Zutaten.

b) Entwerfen Sie eine Funktion `MixDrink( recipe )`, die auf Ihrer Datenstruktur operiert. Geben Sie Pseudocode an. Sie müssen dabei nur Ihre Datenstruktur aktualisieren. Sonstige Funktionen des Roboters müssen Sie nicht berücksichtigen.

c) Wenn ein Kanister gewechselt wird, muss ihre Datenbasis natürlich auch aktualisiert werden. Beschreiben Sie, welche Auswirkungen das Wechseln auf Ihre Datenstruktur hat.

Musterlösung:

a) Die Zutat, die auf dem Display angezeigt werden soll, ist die, die den geringsten Restvorrat vorweist. Zusätzlich sollen die Mengen aller Zutaten im laufenden Betrieb aktualisiert werden können. Dabei werden Zutaten in ihrer Menge immer reduziert. Die klassische Datenstruktur hierfür ist ein adressierbarer *Heap*.

b) Die Funktion `MixDrink`, wie gefordert, hat nur die Aufgabe die vorhandene Zutatenmenge aktuell zu halten.

```
function MixDrink(r : Recipe, q : Queue, d : Display)
    for ingredient i ∈ r do
        q.decreaseKey(i, amount(i))
    end for
    d.show(q.min(), amount(q.min()))
end function
```

c) Beim Wechsel eines Kanisters muss das Element aus dem *Heap* entfernt werden und mit vollem Kanisterwert wieder eingefügt werden. Dabei ist allerdings zu beachten, dass nicht zwingend das aktuell minimale Element gewechselt wird. Daher sollte eine Folge von Operationen durchgefürt werden. Als erstes sollte der Schlüssel der betroffenen Zutat auf $-\infty$ gesetzt werden. Danach kann ein `deleteMin` gefolgt von einem `insert` ausgeführt werden.

Aufgabe 7 (*Implementierung: Dijkstra's Algorithmus*)

Implementieren Sie Dijkstra's Algorithmus in `C++` mit einer adressierbaren Prioritätsliste Ihrer Wahl (https://github.com/) und der Graph-Datenstruktur aus dem KaHIP[1] Framework. Laden Sie das Archiv[2] von unserer Website mit 6 Straßennetzwerken herunter. Das Archiv beinhaltet das komplette Straßennetzwerk der USA, aber auch kleinere gewichtete Teilnetzwerke wie das Straßennetzwerk von Colorado und Florida. Die Netzwerke sind im Metis Format gespeichert. Nutzen Sie die Methoden aus dem KaHIP Framework um die Netzwerke einzulesen.

[1] `http://algo2.iti.kit.edu/kahip`
[2] `http://algo2.iti.kit.edu/schulz/graphs.tar.gz`