Algorithmen und Datenstrukturen
Kapitel 7B: Graphen – Kürzeste Wege

Prof. Dr. Wolfgang Mühlbauer
Fakultät für Informatik
wolfgang.muehlbauer@th-rosenheim.de

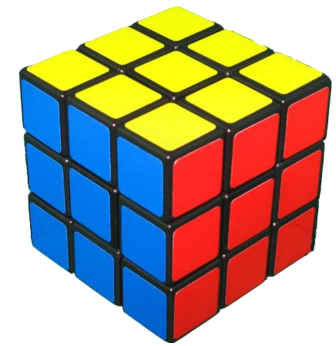
Wintersemester 2019/2020

Übersicht

- Graphen als Datenstruktur
  - Siehe Kapitel 8A

- **Kürzeste Wege**
  - **Definitionen**
  - Algorithmus von Bellman-Ford
  - Algorithmus von Dijkstra
  - ADT: Priority Queue

Kürzeste Wege: Anwendungen



Quelle: [5]

- ***Navigation im Straßenverkehr***
  - Abstraktion: Kanten? Knoten?
  - Welche Eigenschaften haben Kanten?

- ***Rubik‘s Cube [4]***
  - Knoten repräsentieren alle "Konfigurationen" des Würfels
  - Kante falls direkter Übergang zwischen 2 Konfigurationen möglich.

- ***Rechnernetze***
  - Routingalgorithmen, siehe Vorlesung Rechnernetze

- ***Very Large Scale Integration (VLSI)***
  - Entwurf von digitalen Integrated Circuits (ICs)

- Benötigt: *Eigenschaften* von Kanten,
  - Kantengewichte repräsentieren z.B.: Zeit, Kosten, Verlustraten, Entfernungen, etc.

Das "Kürzeste-Wege" Problem

- ***Eingabe:***
  - Gerichteter *Graph G(V,E)*
  - *Gewichtungsfunktion* $w: E \rightarrow \mathbb{R}$
    - Ordnet jeder Kante $\mathrm{e} \in E$ ein Gewicht zu.
    - Negative Kantengewichte sind möglich! Bedeutung?
  - *Startknoten* $s \in V$

Quellcode: EdgeWeightedDiGraph.java

- ***Ausgabe:***
  - Kürzeste Weglänge von *s* zu jedem Knoten in $\mathrm{v} \in V$.
  - Weglänge: Summe aller Kantengewichte des Pfades.

- ***Beobachtungen:***
  - Nur falls alle Kantengewichte 1: Breitensuche ist bereits die Lösung!
  - Es kann mehrere kürzesten Pfade geben.
  - Die kürzesten Pfade zu jedem Knoten ergeben einen Baum.

Kürzeste Wegesuche: Varianten

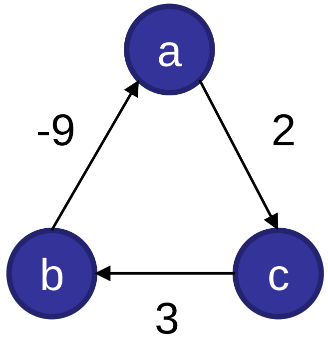
- **Single-source shortest-path (SSSP)**
  - Kürzeste Wege von ***festem*** Startknoten *s* zu ***allen*** Zielknoten
  - Algorithmus: **Dijkstra, Bellman-Ford**
  - Fokus dieser Vorlesung!

- **All-pairs shortest-paths (APSP)**
  - Kürzeste Wege zwischen ***allen*** möglichen Paaren von Start- und Zielknoten
  - Algorithmus: Floyd-Warshall
  - Kein Bestandteil der Vorlesung.

- **Single-destination shortest-path**
  - Kürzester Weg zwischen ***festem*** Start- ***und*** Zielknoten
  - Überraschend: Nicht viel "einfacher" als SSSP
  - Beispiel: A*-Algorithmus
    - (dieser verwendet aber zusätzlich eine Schätzfunktion)

Eingabegraph, Zyklus, negative Gewichte, …

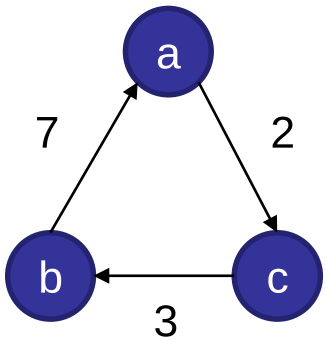
- **Negative Kantengewichte**
  - Prinzipiell erlaubt.
  - Problematisch, falls ***negativer Zyklus!***
    - Beliebig kleine Distanzen durch wiederholtes Durchlaufen des Zyklus'!
  - Negativer Zyklus praktisch nur sinnvoll, wenn dieser vom Startknoten aus gar nicht erreichbar ist.



Negativer Zyklus

- Jeder Teilpfad eines kürzesten Pfades ist selbst ein kürzester Pfad.

- Kann kürzester Pfad ***positiven Zyklus*** enthalten?
  - Nein! Warum?

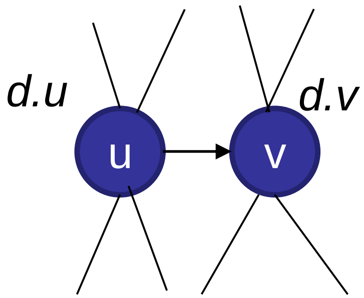


Positiver Zyklus

SSSP Algorithmen: Überblick und Annahmen

Gemeinsamkeiten

- Funktionieren für gerichtete und ungerichtete Graphen.
- Algorithmen berechnen
  - **v.d**: Distanz zum Startknoten.
    - zu Beginn ∞
    - Reduziert sich dann im Laufe des Algorithmus.
  - **v.π**: Vorgänger von *v* auf dem kürzesten Pfad von *s*.
- Inkrementelles Hinzulernen von Informationen
  - Neue Kante *(u, v)* wird mit Gewicht *w* wird entdeckt.
  - Kommt man mit (*u,v)* kürzer zu *v* als über kürzesten Weg, der bislang für *v* bekannt: d.h.: *d.u* + *w* < *d.v ?*



Unterschiede:

- ***Bellman-Ford***: Negative Kantengewichte erlaubt
  - (Negative Zyklen sind nicht erlaubt, aber erkennbar)
- ***Dijkstra***: Setzt positive Kantengewichte voraus!

Übersicht

- Graphen als Datenstruktur
  - Definition
  - Speichern von Graphen: Adjazenzliste und Adjazenzmatrix
  - Durchlaufen von Graphen: Breitensuche
  - Durchlaufen von Graphen: Tiefensuche
  - Topologische Sortierung von gerichteten Graphen
- Kürzeste Wege
  - Definitionen
  - **Algorithmus von Bellman-Ford**
  - Algorithmus von Dijkstra
  - ADT: Priority Queue

Algorithmus von Bellman-Ford

- SSSP = Kürzester Weg vom Startknoten *s* zu ***allen anderen*** Knoten
- ***Erlaubt negative*** Kantengewichte
- ***Erkennt negative Zyklen***, die von s aus erreichbar sind und gibt in diesem Fall `FALSE` zurück.
- Berechnet für alle Knoten
  - ***d.v***: Länge des kürzesten Pfades von *s* zu *v*.
  - ***π.v***: Vorgängerknoten von *v* auf dem kürzesten Weg von *s* zu *v* (*Kürzester-Wege-Baum)*
- ***Asynchrone Version möglich***
  - Keine feste Einteilung in Runden.
  - Jeder Knoten tauscht zu beliebiger Zeit seine aktuellen Entfernungen mit seinen Nachbarn aus (Anwendung: Routingprotokolle wie RIP)

Bellman-Ford Algorithmus

Berechnet die Entfernung aller Knoten vom Startknoten s. Falls negativer Zyklus, Rückgabe "false"

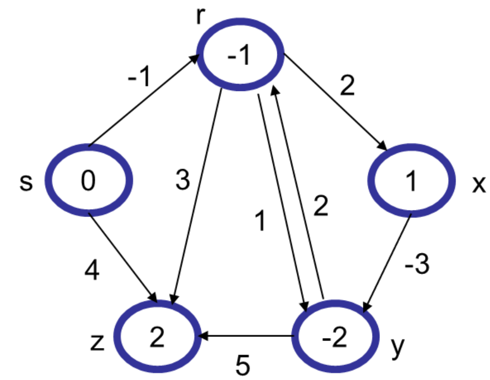
```
BELLMAN-FORD(G, w, s)
    for each vertex v ∈ V
        v.d = ∞
        v.π = NIL
    s.d = 0

    for i = 1 to |V|-1
        for each edge (u,v) ∈ E
            if v.d > u.d + w(u.v)
                v.d = u.d + w(u,v)
                v.π = u

    for each edge (u,v) ∈ E
        if v.d > u.d + w(u.v)
            return FALSE
    return TRUE
```

Quellcode: MyBellmanFord.java

Initialisierung



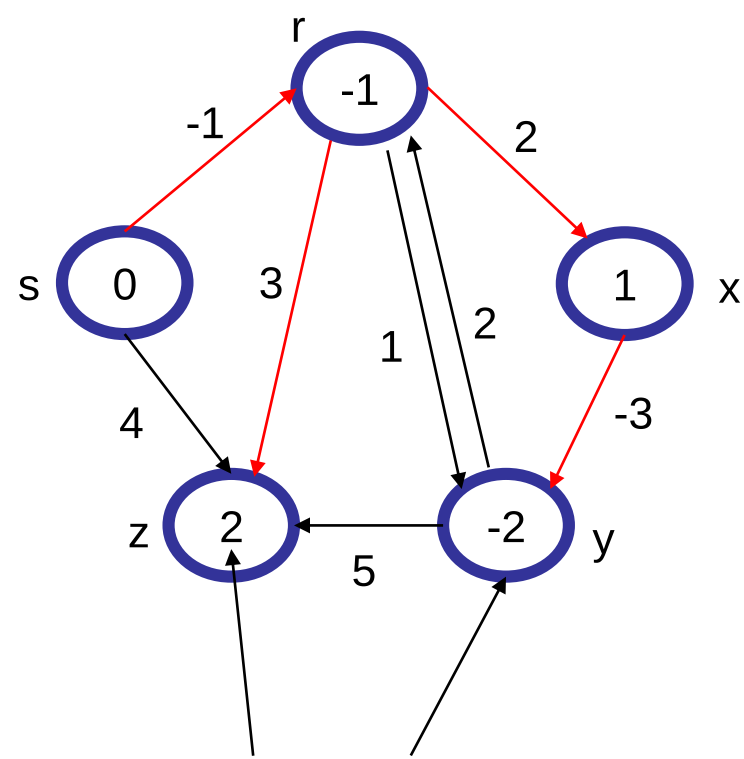
|*V*|-1 ***Runden***, in jeder Runde wird jede Kante einmal besucht.

Falls bei Besuch einer Kante, der Weg über diese Kante kürzer ist als der bislang kürzeste Weg zu *v*, dann passe die Entfernung an.

Falls nach allen Durchläufen noch ein kürzerer Weg gefunden werden kann, muss es einen negativen Zyklus geben.

Bellman-Ford: Übung

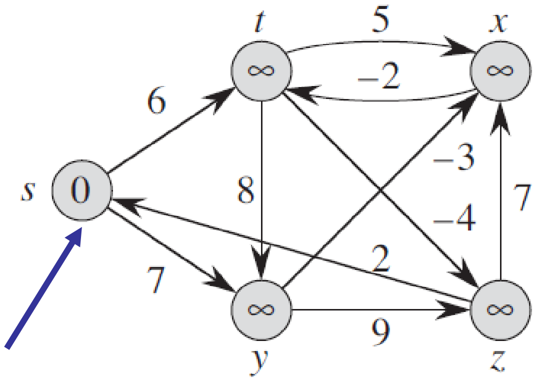
- **Übung**
  - Kürzeste Wege von *s* zu *r, x, y, z* mit Bellman-Ford?
  - Annahme: Kanten (*u,v*) werden in alphabetischer Reihenfolge besucht, d.h. (*s,r*) vor (*s,z*) und (*s,r*) vor (*y,r*).
- **Beobachtung**
  - Wie sich die Distanzwerte ändern, hängt davon ab, in welcher Reihenfolge die Kanten besucht werden.
  - Nach |*V*|-1 Runden hat man aber definitiv das korrekte Ergebnis, (falls keine negativen Zyklen).
  - Begründung: Der längste Pfad in einem Graphen hat nämlich die Länge |*V*|-1.



Kürzeste Entfernungen für jeden Knoten gesucht!

Rote Kanten sind Teil des Kürzeste-Wege-Baumes!

Bellman-Ford: Beispiel



(a)

(b)

(c)

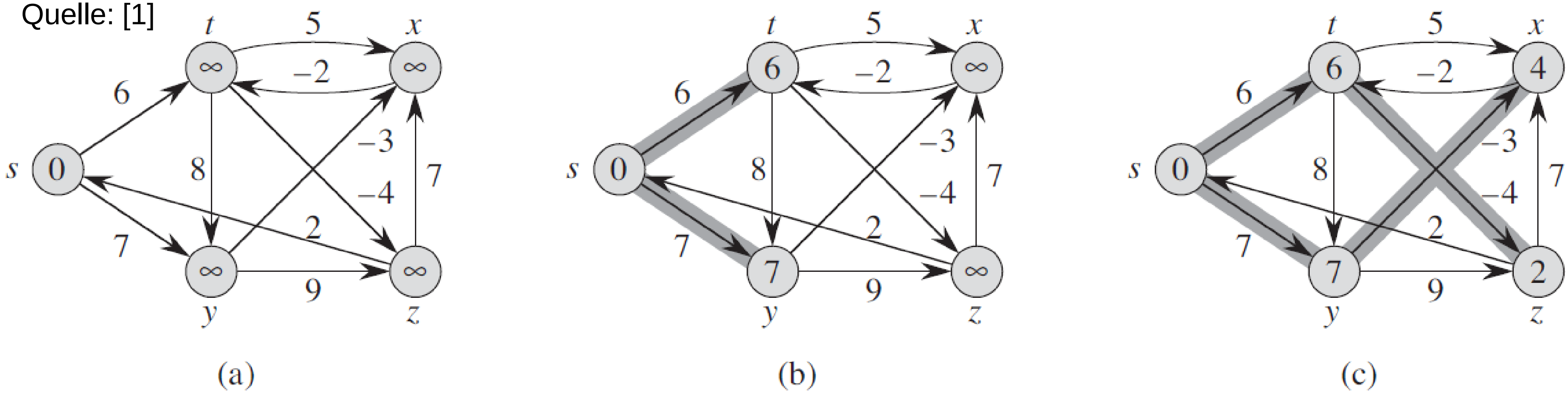
(d)

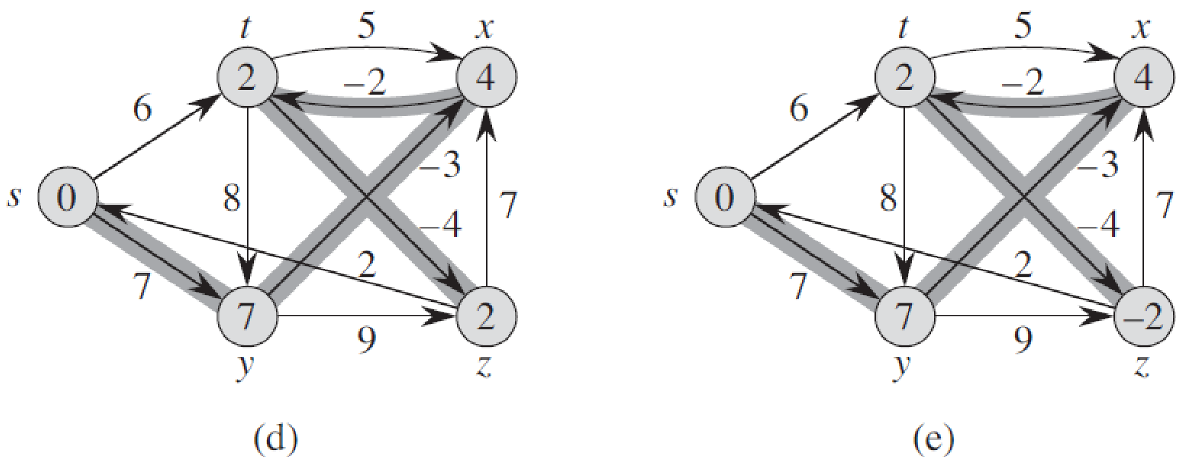
(e)

Annahme: Jede Iteration besucht die Kanten in der Reihenfolge:
(t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (s,t), (s,y).
Vorgängerkanten sind schattiert.

Bellman-Ford: Beispiel

Quelle: [1]





Annahme: Jede Iteration besucht die Kanten in der Reihenfolge:
(t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (s,t), (s,y).
Vorgängerkanten sind schattiert.

Bellman-Ford: Diskussion

- **Laufzeit**: $O(|V| * |E|)$
  - Jede Kante wird $|V|$-mal besucht, siehe Pseudocode
- Algorithmus funktioniert auch in Graphen mit positiven Zyklen
  - Negative Zyklen werden zumindest erkannt!
- ***Parallele, asynchrone*** Version möglich
  - Beliebige Reihenfolge, in der Kanten besucht werden, möglich.
  - Nicht einmal Runden notwendig: Kante kann erneut "besucht" werden, obwohl noch nicht alle anderen Kanten besucht wurden.
  - Falls Algorithmus lange genug läuft: Distanzen konvergieren zum korrekten Ergebnis.
  - Anwendung Routing in Rechnernetzen: Benachbarte Router tauschen periodisch Distanzinformationen miteinander aus.
- Implementierung in Java: Siehe `MyBellmanFord.java`
- Animation
  - https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford/index_en.html

Publikums-Joker: Bellman-Ford

Welche der folgenden Aussagen bzgl. des Bellman-Ford Algorithmus ist ***falsch***?

A. Der formulierte Algorithmus hat grundsätzlich die Laufzeit $O(|V| * |E|)$

B. Selbst bei einer Kette (=lineare Liste), kann der Algorithmus unter Umständen bereits nach 1 Runde alle minimalen Distanzen ermittelt haben.

C. Falls der Graph voll vermascht ist (***jeder*** Knoten ist mit ***jedem*** anderen Knoten benachbart), dann hat der Algorithmus bereits nach 1 Runde die minimalen Distanzen ermittelt.

D. Der Algorithmus kann negative Zyklen erst in der $|V|$.ten Runde zuverlässig erkennen.

Übersicht

- Graphen als Datenstruktur
  - Siehe Kapitel 8A

- Kürzeste Wege
  - Definitionen
  - Algorithmus von Bellman-Ford
  - **Algorithmus von Dijkstra**
  - ADT: Priority Queue

Algorithmus von Dijkstra

- Annahme: Keine negativen Kantengewichte

- ***Starke Ähnlichkeit zur Breitensuche***
  - "Gewichtete" Version der Breitensuche.

- ***Unterschied zur Breitensuche***
  - Verwende ***Priority Queue*** (dt. Vorrangwarteschlange) anstatt *FIFO Queue.*
  - Schlüssel der Priority Queue sind die aktuellen Entfernungen zum Startknoten.

- Unterhalte ***2 Knotenmengen***
  - **S**: Knoten, deren kürzesten Pfade (bzw. Entfernungen) bereits bestimmt sind.
  - **Q**: Priority Queue, die alle restlichen Knoten, d.h. $V \setminus S$, speichert.

- **Priority Queue `Q`** wird vorerst als **Blackbox** verwendet. Annahme:
  - Schneller Zugriff auf den kleinsten Schlüssel.
  - Methode `EXTRACT-MIN():` Entfernt kleinsten Schlüssel und stellt sicher, dass danach wieder das kleinste Element schnell zugreifbar ist.
  - Häufige Implementierung als MinHeap, siehe nächstes Kapitel.

Dijkstra-Algorithmus

Berechnet die Entfernung aller Knoten vom Startknoten s.
Keine Zyklenerkennung!

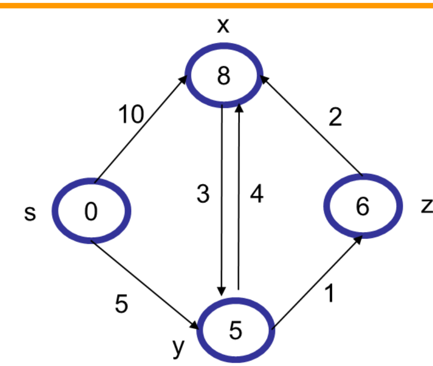
```
DIJKSTRA(G, w, s)
    for each vertex v ∈ V
        v.d = ∞
        v.π = NIL

    s.d = 0
    S = ∅
    Q = V

    while Q not empty
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}

        for each vertex v ∈ G.Adj[u]
            if v.d > u.d + w(u.v)
                v.d = u.d + w(u,v)
                v.π = u
```

Quellcode: MyDijkstra.java



Initialisierung

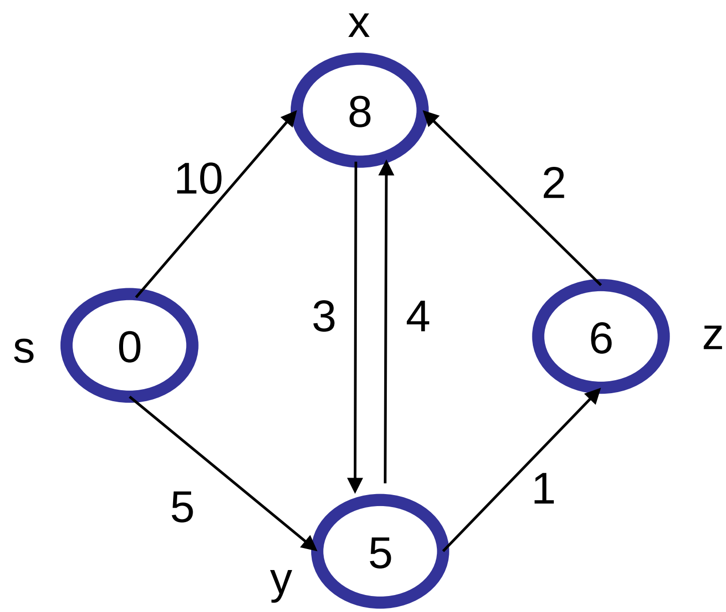
Lege Priority Queue *Q* an, die zu Beginn alle Knoten enthält; als Schlüssel wird *v.d* verwendet; *Q* enthält die Knoten, zu denen kürzeste Entfernung ***noch nicht*** endgültig bekannt ist.

Mache mit Knoten *u* weiter, der aktuell die geringste Distanz zum Startknoten hat (**=*greedy***); entferne Knoten aus Queue

Falls der Weg über Kante (*u*,*v*) kürzer ist als der bislang kürzeste Weg zu *v*, dann passe die Entfernung an.

Dijkstra-Algorithmus: Übung

- **Übung**
  - Kürzeste Wege von *s* zu *x*, *y*, *z* mit Dijkstra?



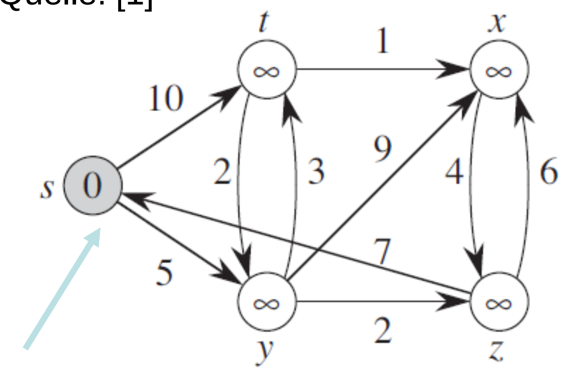
Kürzeste Entfernungen vom Startknoten s gesucht!

- **Beobachtung**
  - Der Algorithmus wählt immer den Knoten mit der aktuell geringsten Distanz aus Q und fügt ihn der Menge *S* hinzu.
  - Man bezeichnet dieses Vorgehen als "**greedy**".
    - Man wählt immer die ***lokal optimale*** Lösung.
    - Man kann beweisen, dass die Gesamtlösung ***global optimal*** ist.
  - Umsetzung: Priority Queue

Gelbe Kanten sind Teil des Kürzeste-Wege-Baumes!

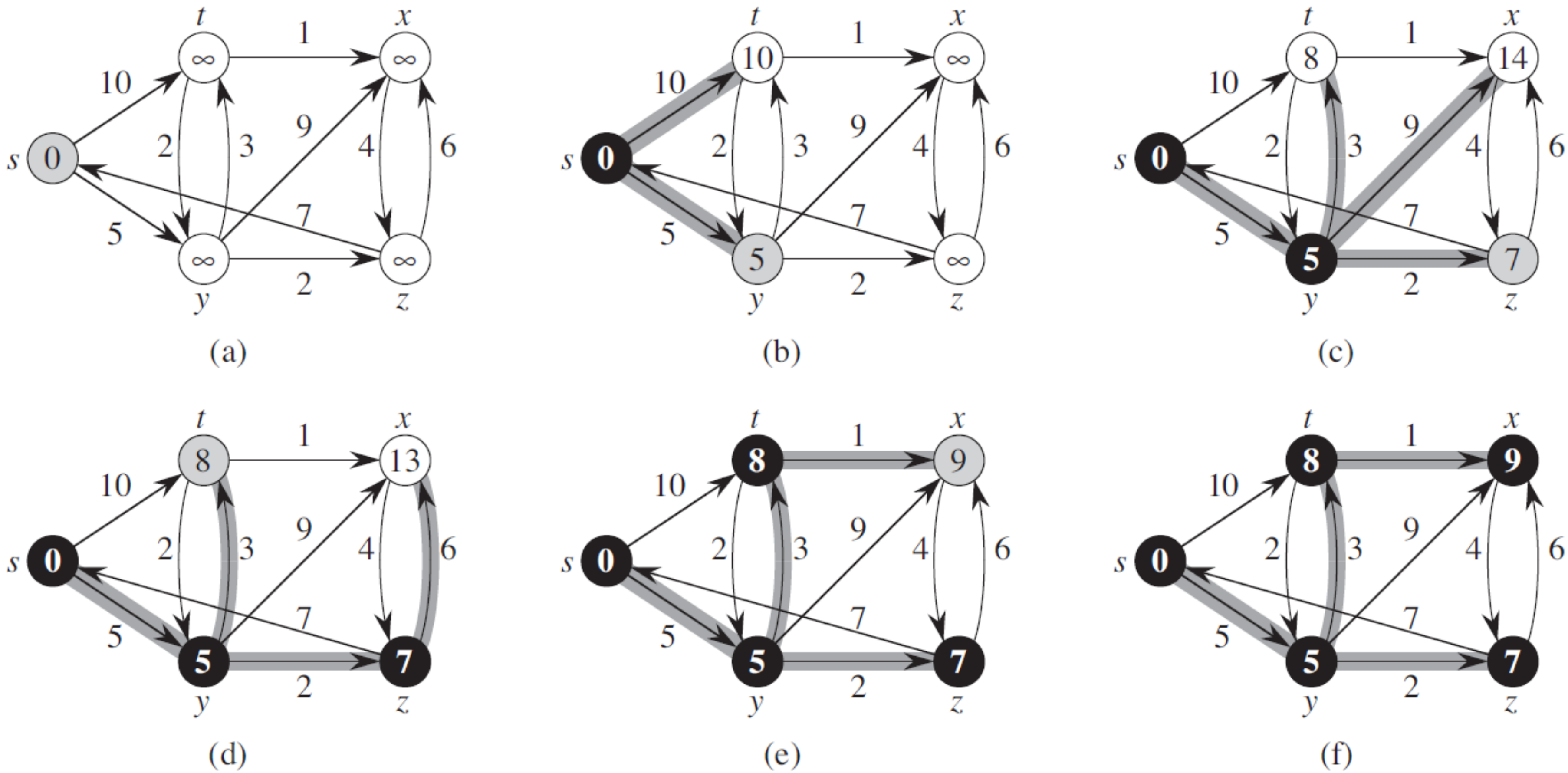
Dijkstra: Beispiel

Quelle: [1]



(a)

Dijkstra: Beispiel



Schattierte Kanten markieren die kürzesten Wege!
Schwarzer Knoten: Kürzeste Entfernung bereits bekannt, in *S* enthalten
Grauer Knoten: Knoten mit kürzester Distanz in Priority Queue

Quelle: [1]

Dijkstra: Laufzeit

Laufzeit falls Priority Queue als MinHeap implementiert ist.

```
DIJKSTRA(G, w, s)
    for each vertex v ∈ V
        v.d = ∞
        v.π = NIL

    s.d = 0
    S = ∅
    Q = V

    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}

        for each vertex v ∈ G.Adj[u]
            if v.d > u.d + w(u.v)
                v.d = u.d + w(u,v)
                v.π = u
```

`INSERT`: |V| Knoten in ***MinHeap*** einfügen:

`EXTRACT-MIN`: Jeder der |V| Knoten wird einmal entfernt:

`DECREASE-KEY`: Hier kann impliziert der Schlüssel *v.d* der Priority Queue verkleinert werden. Die "Ordnung" in der Datenstruktur muss aber wiederhergestellt werden, siehe später.

Implementierung in Java

- Implementierung mit Hilfe der Klasse `PriorityQueue` von Java
  - https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html
  - Verwendet intern einen Heap, siehe nächster Abschnitt.
  - Operationen: `poll`, `remove`, `add`

- `PriorityQueue` benötigt `Comparator` bzw. `Comparable`
  - Beschreibt wie Heap-Elemente verglichen werden.
  - Definiert "Priorität" der Objekte / Knoten
  - Hier: Knoten mit kleinerem *v.d* sind *kleiner*

- Man müsste Code so organisieren, dass man eine eigene Klasse `Node` definiert, die man in `PriorityQueue` ablegen kann.

- Vorsicht beim Ändern von Schlüsselwerten in der Priority Queue!!!
  - Das kann die Heap-Eigenschaft zerstören, siehe DECREASE_KEY!

A* Algorithmus

- Löst nur das "*single-destination shortest-path*" Problem, dieses aber sehr effizient!

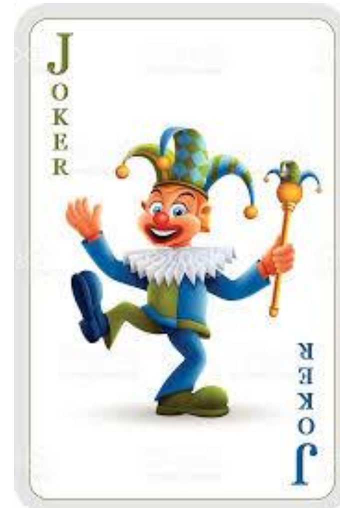
- ***Unterschied zu Dijkstra: Informierter Suchalgorithmus***
  - Besucht zuerst Knoten, die *wahrscheinlich* schnell zum Ziel führen.
  - Andere Priorisierung!

- ***Idee:*** „Distanzmetrik“ bei A* ist die Funktion $f(v) = d(v) + h(v)$
  - $f(v)$ wird in der Priority Queue als Schlüssel verwaltet.
  - $d(v)$ entspricht der ***tatsächlichen*** Distanz *v.d* zum Startknoten, wie bei Dijkstra.
  - $h(v)$ bezeichnet ***geschätzte*** Kosten ("Heuristik") zum Ziel, z.B. Luftlinie zum Ziel!

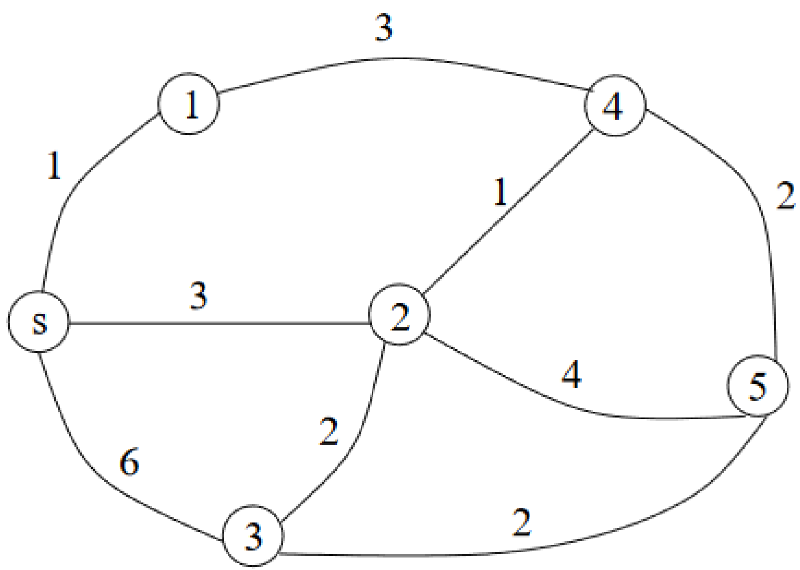
- Details
  - https://de.wikipedia.org/wiki/A*-Algorithmus
  - GDI

Publikums-Joker:

Angenommen Sie verwenden den Dijkstra Algorithmus der Vorlesung auf dem folgenden Graphen, um die kürzesten Wege vom Startknoten *s* zu allen anderen Knoten zu bestimmen. In welcher Reihenfolge werden die finalen Distanzen v.d für die Knoten 1, 2, 3, 4, 5 berechnet?



A. S, 1, 2, 4, 3, 5.
B. S, 1, 2, 3, 4, 5
C. S, 1, 2, 4, 5, 3
D. S, 1, 4, 2, 3, 5



Zusammenfassung

- **Animation für Dijkstra**
  - https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html
  - https://www-m9.ma.tum.de/graph-algorithms/spp-a-star/index_de.html

- **Bellman-Ford**
  - Laufzeit: $O(|V| * |E|)$
  - Annahme: Keine negativen Zyklen!

- **Dijkstra**
  - Laufzeit: $\mathrm{O}(|\mathrm{E}| \log |V|)$ falls Binary MinHeap verwendet wird.
  - Annahme: Alle Kantengewichte sind positiv!

- Hinweis: Falls Graph zyklenfrei ist, SSSP-Problem noch schneller lösbar.
  - Idee: Topologische Sortierung, dann besuche die Knoten in sortierter Reihenfolge → Laufzeit: O(|V| + |E|)

- Hinweis: A*-Algorithmus löst nicht das SSSP Problem
  - Liefert "nur" die kürzeste Route zwischen Start- und Zielknoten.

Übersicht

- Graphen als Datenstruktur
  - Siehe Kapitel 8A

- Kürzeste Wege
  - Definitionen
  - Algorithmus von Bellman-Ford
  - Algorithmus von Dijkstra
  - **ADT: Priority Queue**

ADT: Priority Queue (dt. Vorrangwarteschlange)

- ***Objekte***
  - *Endliche* Menge von Elementen eines bestimmten *Grundtyps.*
  - ***Elemente x*** werden durch einen **Schlüssel (Key) k** identifiziert bzw. lassen sich vergleichen (Java: `Comparable`).

- Im Gegensatz zu normaler Queue kann ***das minimale*** (oder maximale) und nicht das älteste Element schnell gefunden/entfernt werden.

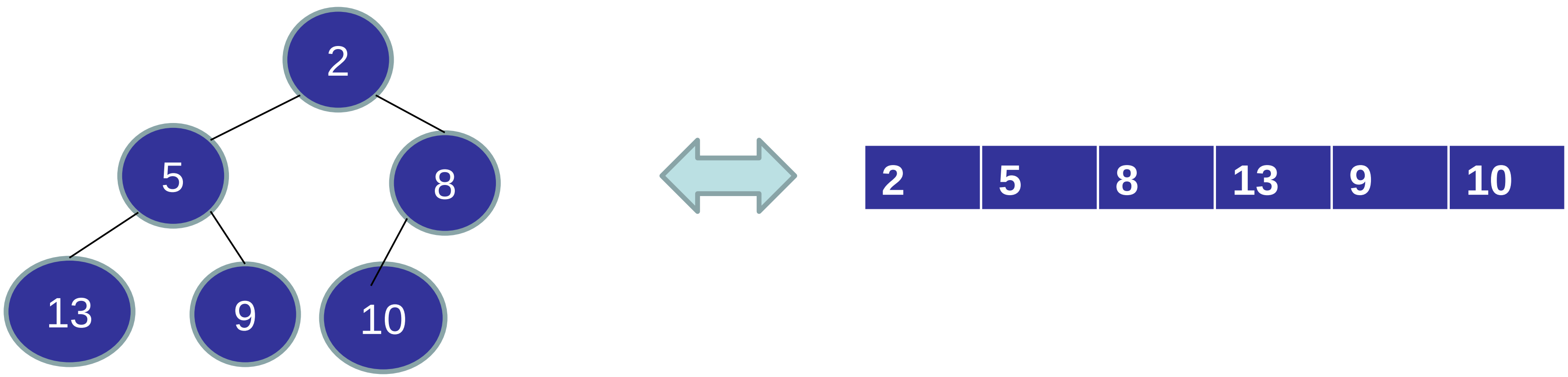
- **Grundoperationen** für Priority Queue *A*
  - `INSERT(k)`
    - Fügt Element mit Schlüssel *k* ein.
  - `MINIMUM()`
    - Gibt das Element mit dem kleinsten Schlüssel zurück ohne es zu entfernen.
  - `EXTRACT-MIN()`
    - Entfernt das Element mit dem kleinsten Schlüssel.
  - `DECREASE-KEY(i, key)`
    - Verkleinert den Schlüssel an Position i des Arrays auf den Wert *key*.

Implementierung einer Priority Queue

- ***Idee: Verwende (Binary) Heap als Datenstruktur***
  - *Minimales Element soll* schnell gefunden werden → ***MinHeap*** (hier der Fall!)
  - Bei maximalem Element ein *MaxHeap* (siehe Heapsort)

- **Wiederholung Definition *Min-Heap*:** Lineare Liste $\left(k_0, k_{1,} \dots, k_{n-1}\right)$, so dass für alle $i = 0, 1, \dots, \frac{n-1}{2}$ gilt: $k_i \leq k_{2i+1}$ und $k_i \leq k_{2i+2}$ sofern $2i < n$ bzw. $2i + 1 < n$
  - Fast immer wird ein Array zur Umsetzung der Linearen Liste bzw. des Heaps verwendet.

Beispiel eines MinHeaps



Heap: Navigation und Operationen

- `PARENT(i)`
  - Index des Elternknotens von *i*
  - **return** $(i - 1)$ `: 2`

- `LEFT (i)`
  - Index des linken Kindknotens
  - **return** `2 * i + 1;`

- `RIGHT (i)`
  - Index des rechten Kindknotens
  - **return** `2 * i + 2;`

- `BUILD-MIN-HEAP (A)`
  - Baut aus beliebigem Heap-Array *A* ein Array, dass der MinHeap-Eigenschaft genügt.

- `HEAPIFY (A, i)`
  - Stellt Heap-Bedingung für den Unterbaum ab Index *i* wieder her, falls diese verletzt ist.

Achtung: Für Dijkstra wird im Gegensatz zu Heapsort ein MinHeap benötigt.

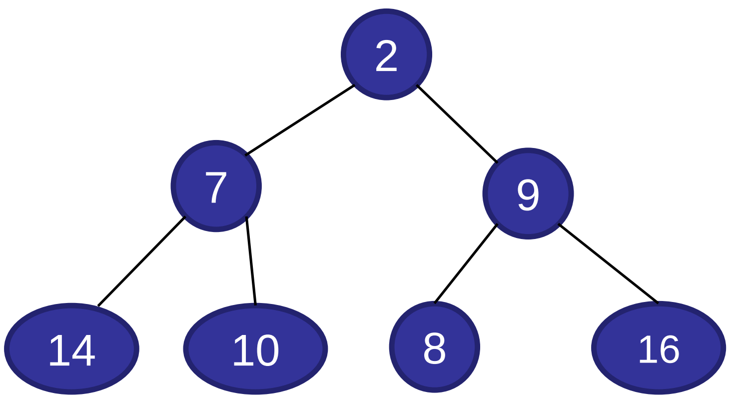
Operation MINIMUM()

- Handelt es sich in der Abbildung um einen ***MinHeap***?

- Wie findet man das Minimum?
  - Das Minimum ist immer die Wurzel!
  - Gib das erste Element des Arrays zurück
  - Annahme im Folgenden: A sei das zum Heap gehörige Array, das die Schlüssel speichert. Der erste Index sei 0.

```
MINIMUM()
1   return A[0]
```

Gibt das Minimum aus dem MinHeap zurück, ohne es zu entfernen.



Operation EXTRACT-MIN

- **Ziel:**
  - Entferne kleinsten Schlüssel. Stelle anschließend Heapeigenschaft wieder her.
- **Idee:**
  - Setze letztes Arrayelement an "Wurzel"
  - Versickern: Stelle die Heap-Bedingung wieder her: MIN-HEAPIFY(), siehe Heapsort.
- **Laufzeit:** *O(log n)*

Stellt Heapeigenschaft wieder für Teilbaum von A[i] wieder her.

```
HEAPIFY(i)
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.length and A[l] < A[i]
        smallest = l
    else
        smallest = i
    if r ≤ A.length and A[r] < smallest
        smallest = r
    if smallest ≠ i
        exchange(A[i],A[smallest])
        MIN-HEAPIFY(smallest)
```

Entfernt das Minimum aus der Priority Queue

```
EXTRACT-MIN()
    if A.length < 1
        error "heap underflow"
    min = A[0]
    A[0] = A[A.length - 1]
    A.length = A.length – 1
    MIN-HEAPIFY(,0)
    return min
```

Quellcode: BinaryHeap.java

Operation DECREASE-KEY(i, key)

- Verkleinere den Schlüssel an Index *i* auf Wert *key*
  - Heap-Eigenschaft kann gestört werden!
  - Benötigt bei Dijkstra

- **Lösung:**
  - Gehe rekursiv den Baum in Richtung Wurzel
  - Vertausche Knoten mit dem Elternknoten so lange bis Schlüssel des Elternknotens kleiner als Schlüssel des Knotens ist.

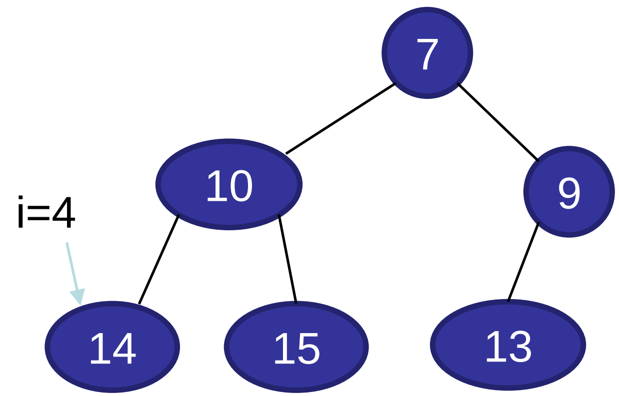
- **Laufzeit:** *O(log n)*

Verkleinere Wert des Schlüssels A[i] in Priority Queue
auf den Wert "key". Heap-Eigenschaft wird danach wiederhergestellt.

```
DECREASE-KEY(i, key)
    if key > A[i]
        error "new key larger than current key"
    A[i] = key
    while i > 0 and A[PARENT(i)] > A[i]
        exchange(A[i], A[PARENT(i)])
        i = PARENT(i)
```

Quellcode: MyPriorityQueue.java

decreaseKey(A,4,6) ?



Operation HEAP-INSERT(k)

- Wie fügt man einen neuen Schlüssel *k* in die Priority Queue ein?

- **Idee:**
  - Füge zunächst "∞" an letzter Arrayposition ein.
  - Rufe `DECREASE-KEY(k)` auf dem letzten Arrayelement auf, um das Element auf den passenden Schlüssel k zu setzen.
  - "Nach oben spülen" anstatt "Versickern".

- **Laufzeit:** *O(log n)*

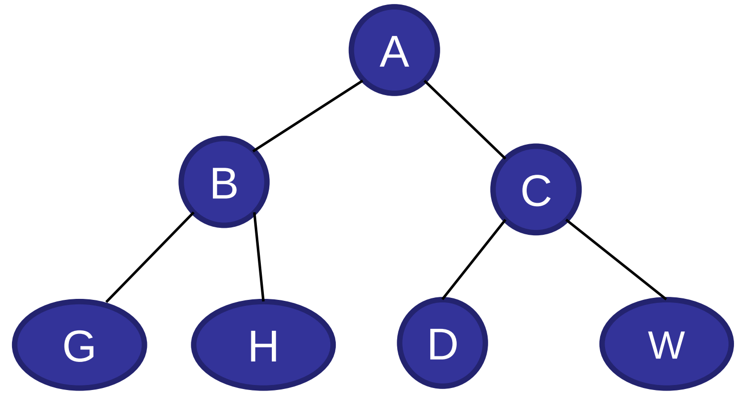
Füge einen neuen Wert "key" in die Priority Queue ein.

```
HEAP-INSERT(key)
    A.length = A.length + 1
    A[A.length - 1] = "∞"
    HEAP-DECREASE-KEY(A.length-1, key)
```

Quellcode:
MyPriorityQueue.java

Exkurs: MyPriorityQueue<Key extends Comparable<Key>>

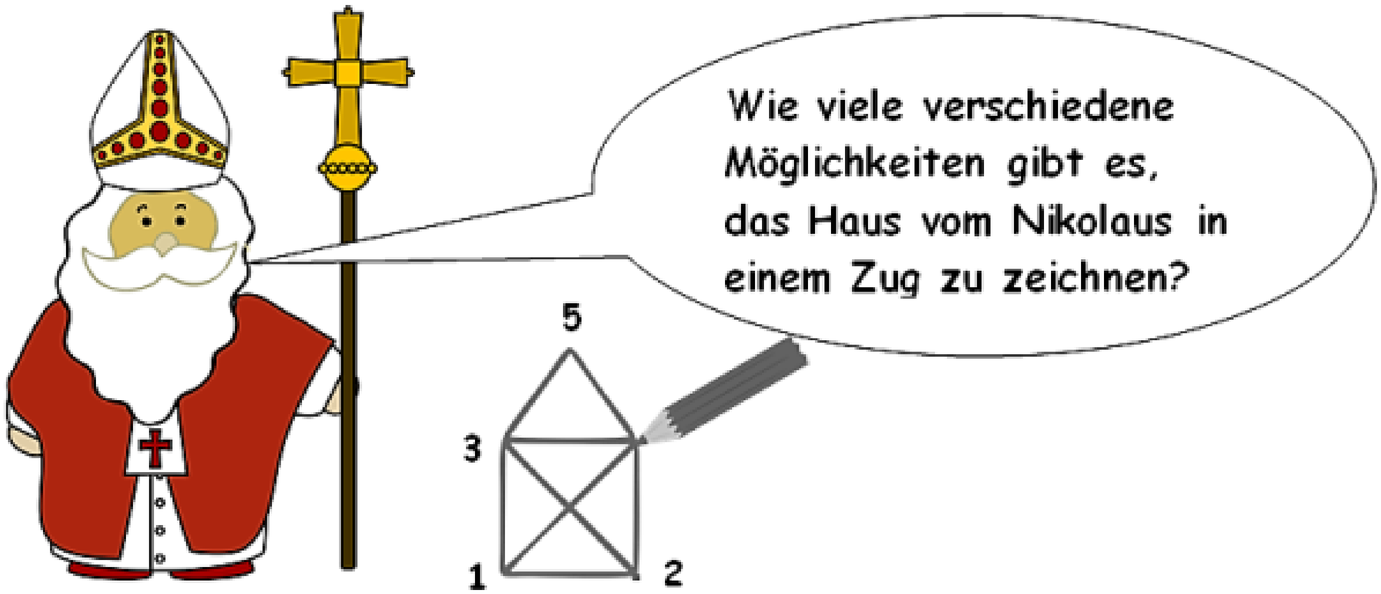
- Benutzer der Priority Queue (PQ) vergibt **Handle/Index,** um später schnell zu eingetragen Schlüssel in Datenstruktur zu gelangen.
  - `public void insert(int i, Key key) {`
  - `i` ist das Handle, um später schnell zu Schlüssel zu navigieren.
- **`keys`**
  - Speichert (beliebige) Objekte, Priorisierung über `Comparable`.
  - **Handle** == Index in Array keys
- **`pq`**
  - Eigentliches Heaparray, speichert **Handles!!!**
  - `pq[0]=2` bedeutet, dass Handle 2 (== 'A') an Wurzel
- **`heapPos`**
  - Gibt an, an welcher Position in pq das Handle im Heaparray steht.
  - heapPos[3] = 4 bedeutet, dass Handle 3 (=='H') an Position 4 des Heaparrays steht.



Array pq	2	5	7	0	3	4	8	frei	frei	frei
Array heapPos	3	frei	0	4	5	1	frei	2	6	frei
Array keys	G	frei	A	H	D	B	frei	C	W	frei
"Handles"	0	1	2	3	4	5	6	7	8	9

Gehören zusammen

Haus des Nikolaus



Tipps:

- Welche Anzahl vermutest du?
- Wie kannst du deine Vermutung überprüfen?
- An welchen Eckpunkten kann man beginnen?

Frohe Weihnachten und einen guten Start ins neue Jahr!

Zusammenfassung

- Mittels eines Binary MinHeaps lassen sich alle Operationen in O(log *n*) implementieren!

- Damit hat Dijkstra mit Binary MinHeap die asymptotische Laufzeit: O(|*E*| * log |*V*|)

- Weitere Implementierungsmöglichkeiten einer Priority Queue
  - Binomial Heap
  - Fibonacci Heap

- Animation: MinHeap
  - https://www.cs.usfca.edu/~galles/visualization/Heap.html

- JGraphT Library:
  - http://jgrapht.org/javadoc/org/jgrapht/alg/BellmanFordShortestPath.html
  - http://jgrapht.org/javadoc/org/jgrapht/alg/DijkstraShortestPath.html

Quellenverzeichnis

[1] Cormen, Leiserson, Rivest and Stein. *Introduction to Algorithms*, Third Edition, The MIT Press, 2009.

[2] Ottmann, Widmayer. *Algorithmen und Datenstrukturen*, Kapitel 9, 5. Auflage, Spektrum Akademischer Verlag, 2012.

[3] Quelle: http://people.seas.harvard.edu/~babis/amazing.html (abgerufen am 03.12.2016)

[4] Rubik's Cube, *Introduction to Algorithms*, https://courses.csail.mit.edu/6.006/fall11/rec/rec16.pdf (abgerufen am 11.12.2016)

[5] https://commons.wikimedia.org/wiki/File%3ARubiks-Cube.gif (abgerufen am 11.12.2016)