ALGORITHMEN UND DATENSTRUKTUREN

ÜBUNG 10: TOPOLOGISCHES SORTIEREN & GRAPHENSUCHE

Eric Kunze
eric.kunze@tu-dresden.de

TU Dresden, 13. Dezember 2021

letzte Änderung:
16.12.2021, 22:36

Topologisches Sortieren
Implementierung

TOPOLOGISCHES SORTIEREN

Gegeben sei ein gerichteter, azyklischer Graph $G = (V, E)$. Eine **topologische Sortierung** von G ist eine *bijektive* Abbildung $\mathrm{ord}\colon V \to \{1, \ldots, |V|\}$, sodass für alle $v, v' \in V$ mit $(v, v') \in E$ die Relation $\mathrm{ord}(v) < \mathrm{ord}(v')$ gilt.

Anschauung: $\mathrm{ord}(v) = n \quad \rightsquigarrow \quad$ Knoten v wird als n-tes Element gewählt (erhält Sortierungsnummer n)

TOPOLOGISCHES SORTIEREN

Gegeben sei ein gerichteter, azyklischer Graph $G = (V, E)$. Eine **topologische Sortierung** von G ist eine *bijektive* Abbildung $\mathrm{ord}\colon V \to \{1, \ldots, |V|\}$, sodass für alle $v, v' \in V$ mit $(v, v') \in E$ die Relation $\mathrm{ord}(v) < \mathrm{ord}(v')$ gilt.

Anschauung: $\mathrm{ord}(v) = n \quad \rightsquigarrow \quad$ Knoten v wird als n-tes Element gewählt (erhält Sortierungsnummer n)

Algorithmus:

`while` (Elemente übrig)

- wähle Element v ohne Vorgänger
- dekrementiere Anzahl der Vorgänger in den Nachfolgern von v
- füge v der Ausgabeliste hinzu
- lösche v aus G

KODIERUNGEN

Kanten:

```
struct Edge { int from, to; };
```

Bsp.: Kante $e = (3, 5)$

```
struct Edge e = {3,5}
e.from == 3
e.to   == 5
```

KODIERUNGEN

Kanten:

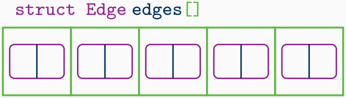
```
struct Edge { int from, to; };
```

Bsp.: Kante $e = (3, 5)$

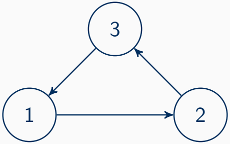
```
struct Edge e = {3,5}
e.from == 3
e.to   == 5
```

Graph: Liste von Kanten

$\rightsquigarrow$ `struct Edge edges[];`



Bsp.: Graph mit Kantenmenge $E = \{(1,2), (2,3), (3,1)\}$



$\rightsquigarrow$
```
struct Edge edges[] =
        {{1,2}, {2,3}, {3,1}};
```

KODIERUNGEN

Kanten:

```
struct Edge { int from, to; };
```

Bsp.: Kante $e = (3, 5)$

```
struct Edge e = {3,5}
e.from == 3
e.to   == 5
```

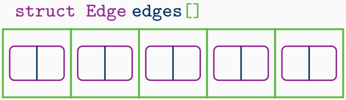
Abbildung ord: Array `int ord[]`

mit $\mathrm{ord}(v) = j \Leftrightarrow$ `ord[v] = j`

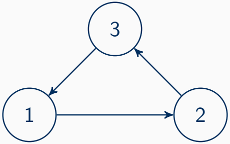
- initialisiert mit -1

Graph: Liste von Kanten

$\rightsquigarrow$ `struct Edge edges[];`



Bsp.: Graph mit Kantenmenge $E = \{(1,2),(2,3),(3,1)\}$



$\rightsquigarrow$
```
struct Edge edges[] =
        {{1,2}, {2,3}, {3,1}};
```

EIN ALTERNATIVER ALGORITHMUS

`while` (Elemente übrig)

- wähle Element v ohne Vorgänger
- dekrementiere Anzahl der Vorgänger in den Nachfolgern von v
- füge v der Ausgabeliste hinzu
- lösche v aus G

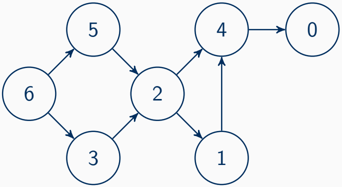
`for` jede Sortierungsnummer j

- `for` jeden Knoten v
  - teste ob v gewählt werden kann, d.h. noch nicht platziert ist und eingehende Kanten bereits platziert
- falls v gewählt werden darf, setze `ord[v] = j`

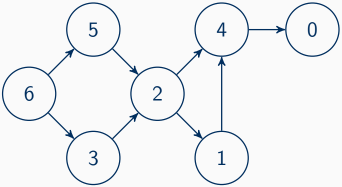
AUFGABE 1

```
void topsort(int n, int e, struct Edge edges[], int ord[]) {
  int j = 1, node, edge, ok;
  while (j <= n)  {
    for (node = 0; node < n; ++node) {
      if (ord[node] == -1) {
        ok = 1;
        for (edge = 0; edge < e; ++edge) {
          if (edges[edge].to == node &&
              ord[edges[edge].from] == -1)
            ok = 0;
        }
        if (ok) {
          ord[node] = j;
          j++;
          break;
        }
      }
    }
  }
}
```

EIN BEISPIEL

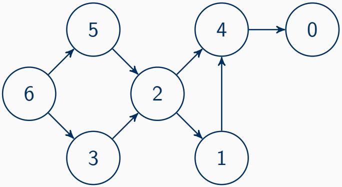


EIN BEISPIEL



Knoten 6 bekommt Nummer 1.
Knoten 3 bekommt Nummer 2.
Knoten 5 bekommt Nummer 3.
Knoten 2 bekommt Nummer 4.
Knoten 1 bekommt Nummer 5.
Knoten 4 bekommt Nummer 6.
Knoten 0 bekommt Nummer 7.

EIN BEISPIEL



Knoten 6 bekommt Nummer 1.
Knoten 3 bekommt Nummer 2.
Knoten 5 bekommt Nummer 3.
Knoten 2 bekommt Nummer 4.
Knoten 1 bekommt Nummer 5.
Knoten 4 bekommt Nummer 6.
Knoten 0 bekommt Nummer 7.

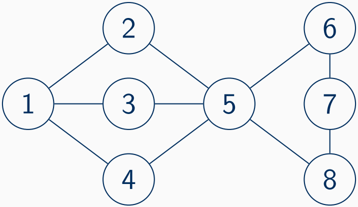
$v =$	0	1	2	3	4	5	6
$\mathrm{ord}(v) =$	7	5	4	2	6	3	1

```
ord = [7, 5, 4, 2, 6, 3, 1]
```

Breiten- und Tiefensuche

SUCHVERFAHREN

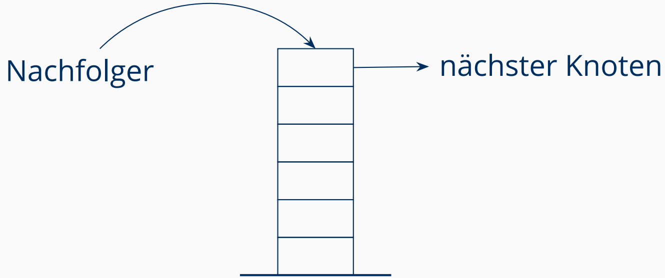
- Ziel: Finden eines Knotens mit bestimmter Beschriftung in einem Graphen
- hier: uninformierte Suche mit Tiefen- und Breitensuche



TIEFENSUCHE

- gehe in die Tiefe:
  „entdecke erst Kinder, dann Geschwister“
- Datenstruktur: Keller
- Nachfolger werden *oben* auf den Keller gelegt
- nächster Knoten wird *oben* vom Keller genommen

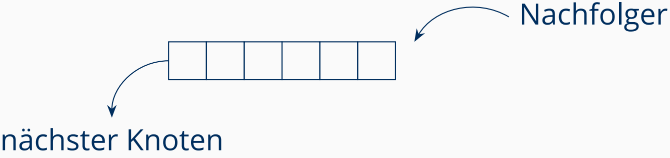
Keller:



BREITENSUCHE

- gehe in die Breite:
  „entdecke erst Geschwister, dann Kinder“
- Datenstruktur: Warteschlange
- Nachfolger stellen sich *hinten* an
- nächster Knoten wird von *vorn* genommen

Warteschlange:



VERALLGEMEINERTE GRAPHENSUCHE

Beobachtung: Suche läuft ähnlich ab

- Operation 1: Lesen des nächsten Knotens `READ`
- Operation 2: Löschen des gewählten Knotens `REMOVE`
- Operation 3: Hinzufügen eines Nachfolgerknotens `INSERT`
- Operation 4: Leerheit der Datenstruktur `EMPTY`

VERALLGEMEINERTE GRAPHENSUCHE

Beobachtung: Suche läuft ähnlich ab

- Operation 1: Lesen des nächsten Knotens `READ`
- Operation 2: Löschen des gewählten Knotens `REMOVE`
- Operation 3: Hinzufügen eines Nachfolgerknotens `INSERT`
- Operation 4: Leerheit der Datenstruktur `EMPTY`

	`STORAGE`	`READ`	`REMOVE`	`INSERT`	`EMPTY`
Tiefensuche	Keller	`top`	`pop`	`push`	`empty`
Breitensuche	Queue	`head`	`dequeue`	`enqueue`	`nil`

VERALLGEMEINERTE GRAPHENSUCHE

Beobachtung: Suche läuft ähnlich ab

- Operation 1: Lesen des nächsten Knotens `READ`
- Operation 2: Löschen des gewählten Knotens `REMOVE`
- Operation 3: Hinzufügen eines Nachfolgerknotens `INSERT`
- Operation 4: Leerheit der Datenstruktur `EMPTY`

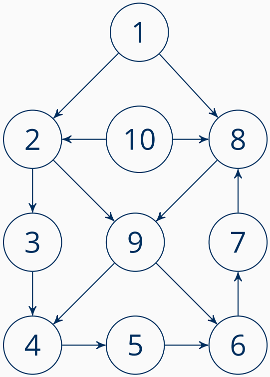
	`STORAGE`	`READ`	`REMOVE`	`INSERT`	`EMPTY`
Tiefensuche	Keller	`top`	`pop`	`push`	`empty`
Breitensuche	Queue	`head`	`dequeue`	`enqueue`	`nil`

weitere Möglichkeit für `STORAGE`: **Prioritätswarteschlange**

(vgl. Übung 11, Dijkstra-Algorithmus für kürzeste Wege in gewichteten Graphen)

- Wahl des nächsten Elementes anhand eines Prioritätswertes
- Vorstellung: *geordnete* Warteschlange

AUFGABE 2



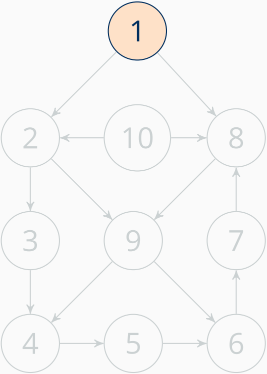
(a) **Tiefensuche**:
3 verschiedene
depth-first trees

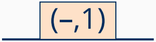
(b) **Breitensuche**:
3 verschiedene
breadth-first trees

Tiefensuche

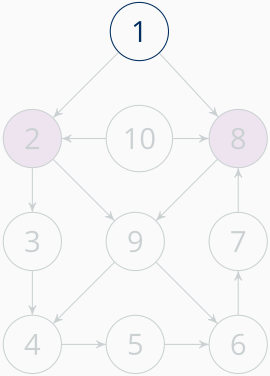
$\boxed{(-,1)}$

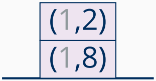
Tiefensuche



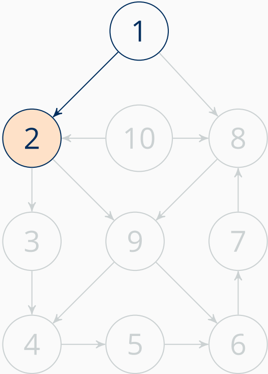


Tiefensuche



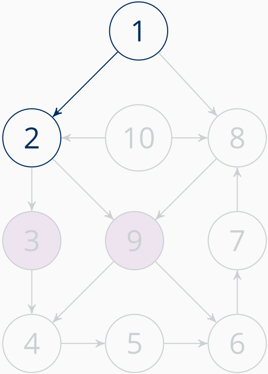


Tiefensuche



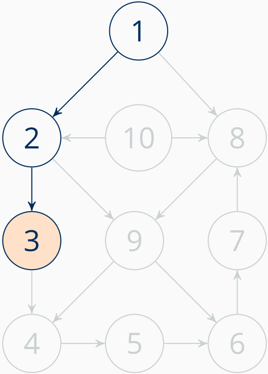
(1,2)
(1,8)

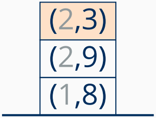
Tiefensuche



(2,3)
(2,9)
(1,8)

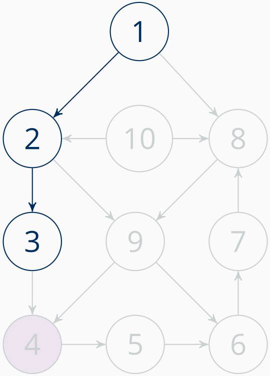
Tiefensuche





AUFGABE 2 — TEIL (A)

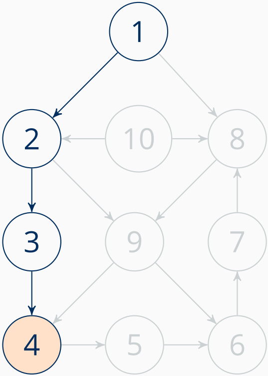
Tiefensuche



(3,4)
(2,9)
(1,8)

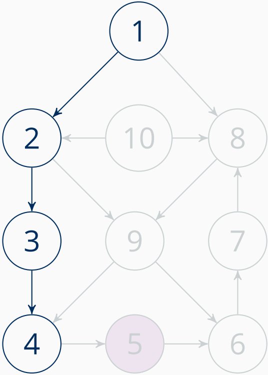
AUFGABE 2 — TEIL (A)

Tiefensuche



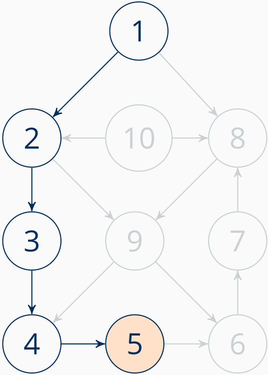
(3,4)
(2,9)
(1,8)

Tiefensuche



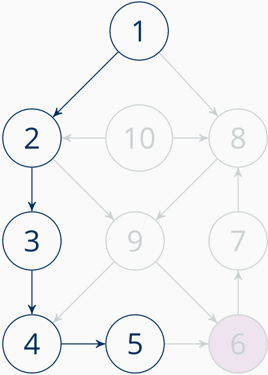
(4,5)
(2,9)
(1,8)

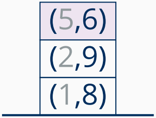
Tiefensuche



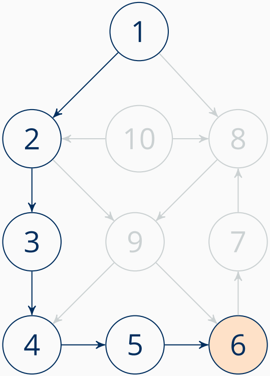
(4,5)
(2,9)
(1,8)

Tiefensuche



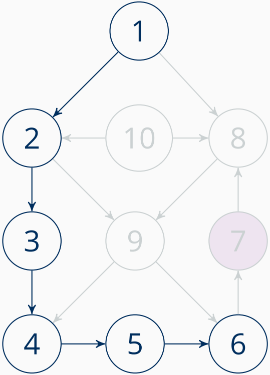


Tiefensuche



(5,6)
(2,9)
(1,8)

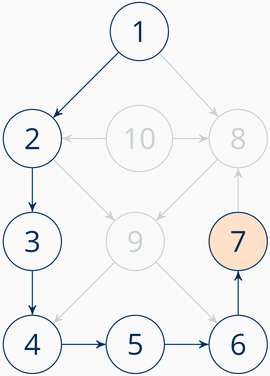
Tiefensuche



(6,7)
(2,9)
(1,8)

AUFGABE 2 — TEIL (A)

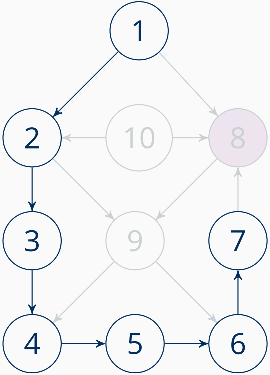
Tiefensuche

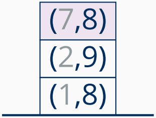


(6,7)
(2,9)
(1,8)

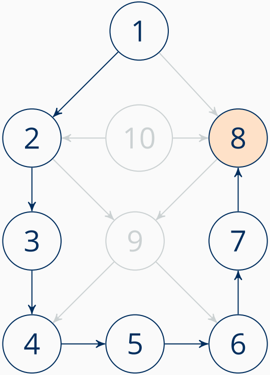
AUFGABE 2 — TEIL (A)

Tiefensuche



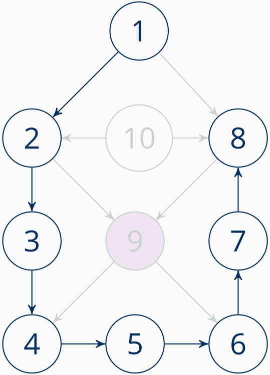


Tiefensuche



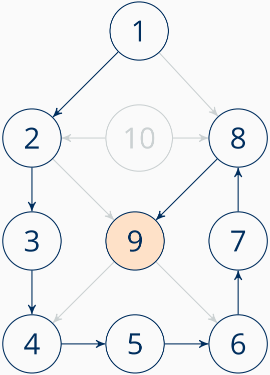
(7,8)
(2,9)
(1,8)

Tiefensuche



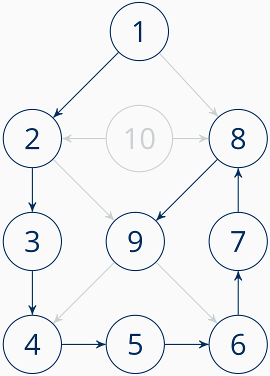
(8,9)
(2,9)

Tiefensuche



(8,9)
(2,9)

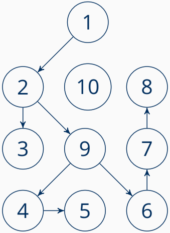
Tiefensuche



AUFGABE 2 — TEIL (A)

Tiefensuche

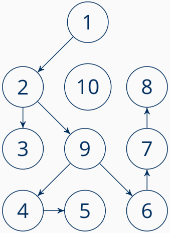
Es gibt 5 verschiedene depth-first-trees, die weiteren DFTs sind:

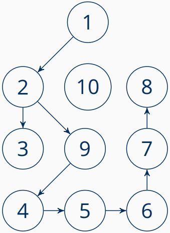


AUFGABE 2 — TEIL (A)

Tiefensuche

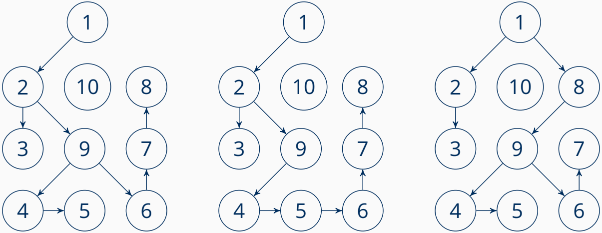
Es gibt 5 verschiedene depth-first-trees, die weiteren DFTs sind:





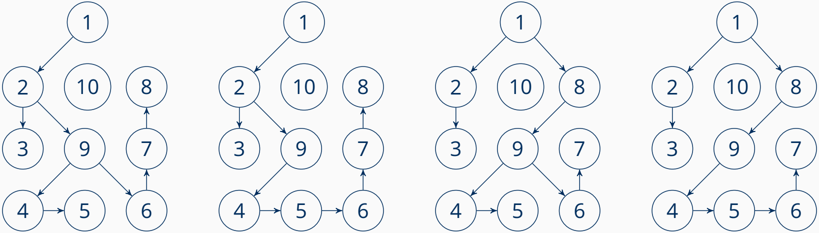
Tiefensuche

Es gibt 5 verschiedene depth-first-trees, die weiteren DFTs sind:



Tiefensuche

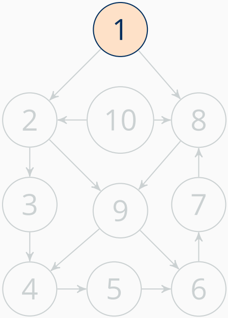
Es gibt 5 verschiedene depth-first-trees, die weiteren DFTs sind:



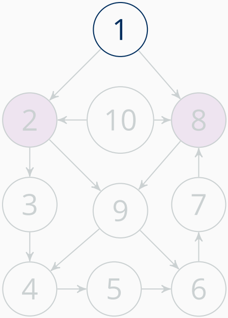
AUFGABE 2 — TEIL (B)

Breitensuche

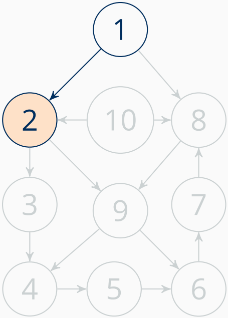
Breitensuche



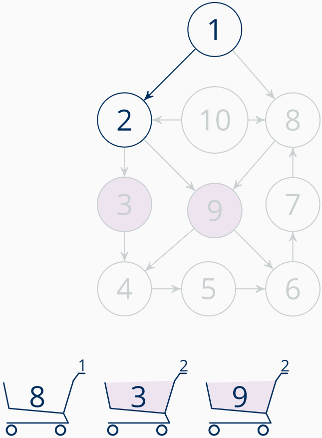
Breitensuche



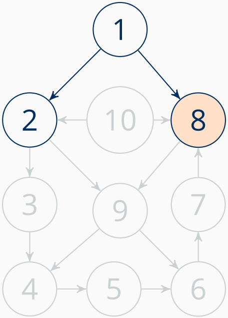
Breitensuche



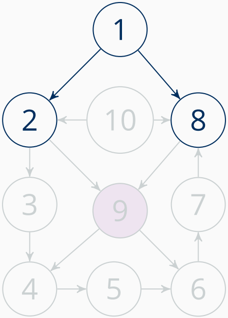
Breitensuche



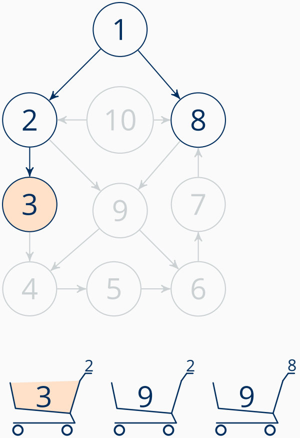
Breitensuche



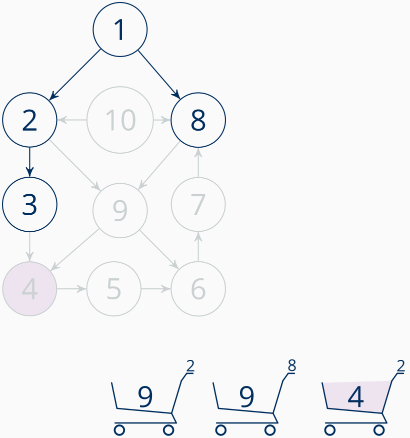
Breitensuche



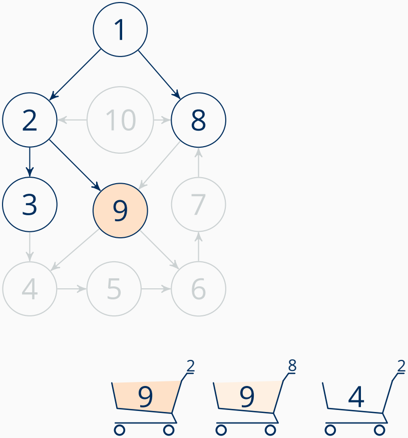
Breitensuche



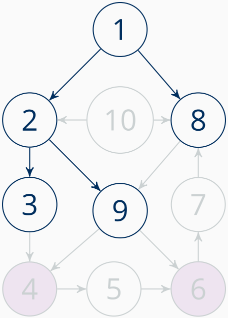
Breitensuche



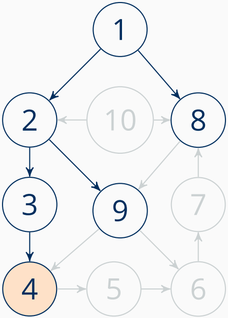
Breitensuche



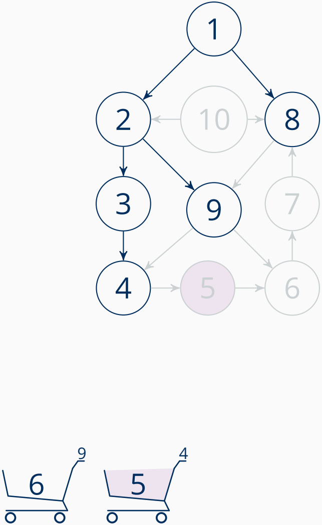
Breitensuche



Breitensuche

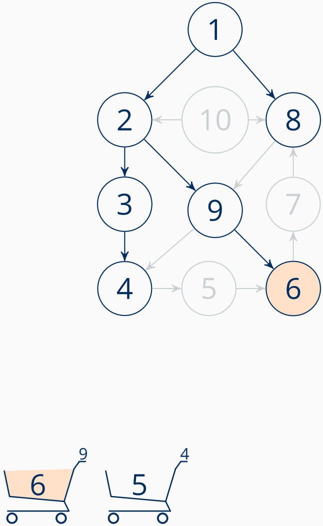


Breitensuche

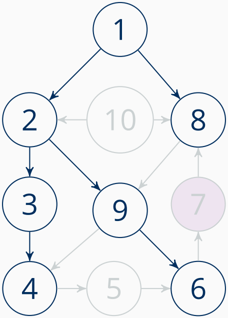


AUFGABE 2 — TEIL (B)

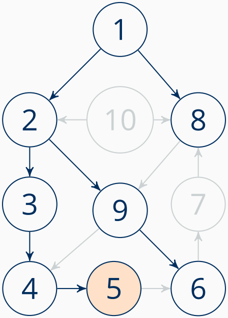
Breitensuche



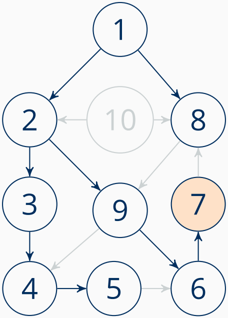
Breitensuche



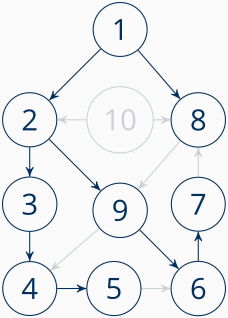
Breitensuche



Breitensuche



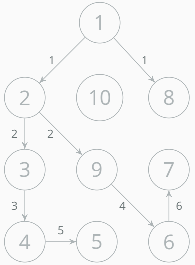
Breitensuche



AUFGABE 2 — TEIL (B)

Breitensuche

Es gibt 5 verschiedene breadth-first-trees, zwei weitere sind z.B.:

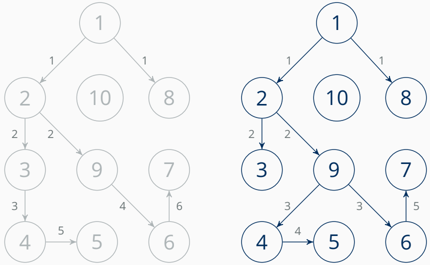


... die zwei weiteren Varianten einer möglichen Warteschlange liefern Bäume, die wir schon gefunden haben. Effektiv gibt es also nur diese drei BFTs.

AUFGABE 2 — TEIL (B)

Breitensuche

Es gibt 5 verschiedene breadth-first-trees, zwei weitere sind z.B.:

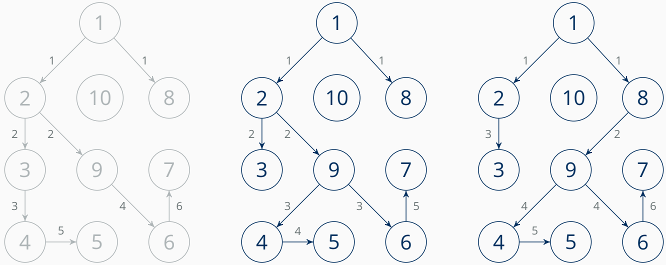


... die zwei weiteren Varianten einer möglichen Warteschlange liefern Bäume, die wir schon gefunden haben. Effektiv gibt es also nur diese drei BFTs.

AUFGABE 2 — TEIL (B)

Breitensuche

Es gibt 5 verschiedene breadth-first-trees, zwei weitere sind z.B.:



... die zwei weiteren Varianten einer möglichen Warteschlange liefern Bäume, die wir schon gefunden haben. Effektiv gibt es also nur diese drei BFTs.