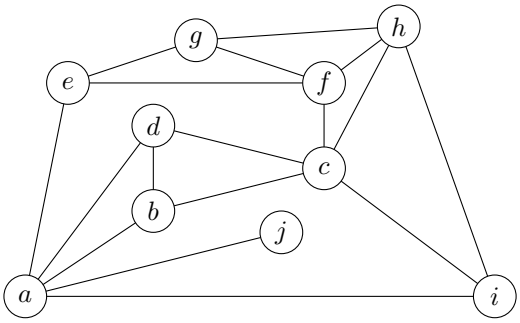
Aufgabe 1.

(a) Führen Sie auf dem nachfolgenden Graphen die Breiten- und Tiefensuche entsprechend den Algorithmen in den Vorlesungsfolien durch. Geben Sie dabei jeweils eine gültige Reihenfolge an, in der die Knoten besucht werden.

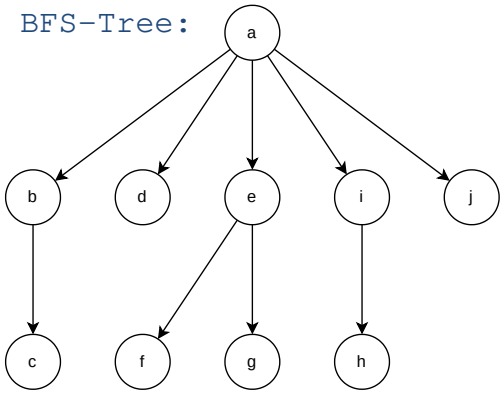
Zeichnen Sie zusätzlich den Breiten- und Tiefensuchbaum. Eine Kante $v \to w$ in diesen Bäumen drückt aus, dass Knoten w von Knoten v aus entdeckt wurde.

Verwenden Sie jeweils a als Startknoten. Haben Sie die Wahl zwischen mehreren Knoten, gehen Sie alphabetisch vor.

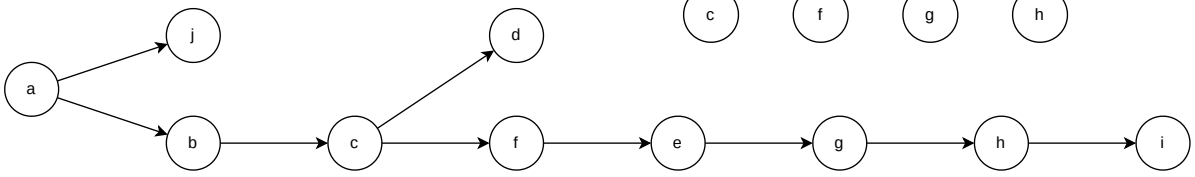


Reihenfolge:
BFS: a,b,d,e,i,j,c,f,g,h
DFS: a,b,c,d,f,e,g,h,i,j

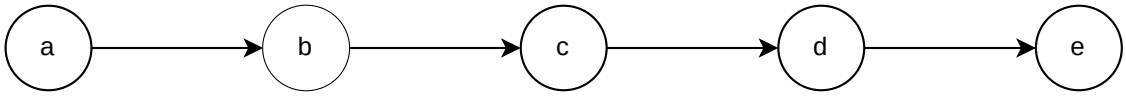
BFS-Tree:



DFS-Tree:



(b) Geben Sie einen zusammenhängenden Graphen mit Knoten a, b, c, d, e an, in welchem eine Breiten- und eine Tiefensuche mit Startpunkt a die Knoten zwingend in der gleichen Reihenfolge besucht.

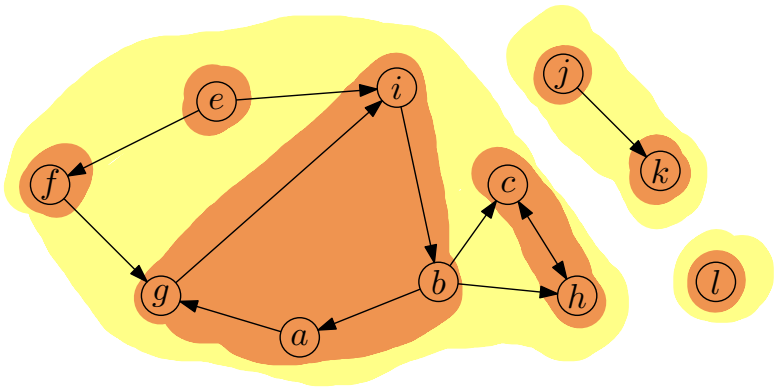


```
BFS(G,s):
Discovered[s] ← true
Discovered[v] ← false für alle anderen Knoten v ∈ V
Q ← {s}
while Q ist nicht leer
    Entferne ersten Knoten u aus Q
    Führe Operation auf u aus (z.B. Ausgabe)
    foreach Kante (u,v) inzident zu u
        if  !Discovered[v]
            Discovered[v] ← true
            Füge v zu Q hinzu
```

```
DFS(G,s):
Discovered[v] ← false für alle Knoten v ∈ V
DFS1(G,s)

DFS1(G,u):
Discovered[u] ← true
Führe Operation auf u aus (z.B. Ausgabe)
foreach Kante (u,v) inzident zu u
    if  !Discovered[v]
        DFS1(G,v)
```

Aufgabe 2. Lösen Sie folgende Unteraufgaben zum Thema Zusammenhangskomponenten.

(a) Bestimmen Sie die starken und schwachen Zusammenhangskomponenten des folgenden Graphen.



(b) Wir wollen zeigen, dass die schwachen Zusammenhangskomponenten die Knoten des Graphen partitionieren. Begründen Sie dafür die folgenden beiden Aussagen.

- Jeder Knoten ist in mindestens einer schwachen Zusammenhangskomponente.
- Jeder Knoten ist in höchstens einer schwachen Zusammenhangskomponente.

Für Interessierte: Die gleichen Aussagen gelten ebenfalls für die starken Zusammenhangskomponenten.

Jeder Knoten steht mit sich selbst in schwachem Zusammenhang, daher ist er in mindestens einer schwachen Zusammenhangskomponente.

Nehmen wir an ein Knoten wäre in zwei schwachen Zusammenhangskomponenten. Dann gäbe es für beide Zusammenhangskomponenten einen Zusammenhang mit dem Knoten und damit auch zwischeneinander. Damit wären die Zusammenhangskomponenten nicht mehr der maximal zusammenhängende Teilgraph (weil es einen größeren gibt).

(c) Entwerfen Sie einen Algorithmus, der die schwachen Zusammenhangskomponenten eines gerichteten Graphen berechnet.

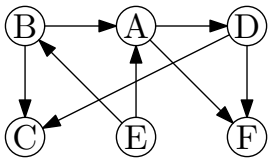
Hinweis: Eine Laufzeit von $O(|V| + |E|)$ ist möglich.

```
# Graph = (V,E)
# Edge = (f,t)

BFSNUM(Graph G):
    # Mache Graph ungerichtet und merke alle hinzugefügten Kanten
    Edge[] Added = new Egde[0]
    foreach (Kante (u,v) ∈ G.E):
        Edge e = (v,u)
        if !G.E.contains(e):
            G.E.add(e)
            Added.add(e)

    Graph[] Result = new Graph[0]
    bool[] Discovered = new bool[|V|]
    Discovered[v] = false für alle Knoten v ∈ G.V

    while (Mindestens ein Element von Discovered ist false):
        Graph R = new Graph()
        Node s = Erster Knoten v von V mit Discovered[v] = false
        Discovered[s] = true
        Queue Q = {s}
        while (!Q.empty()):
            Node v = Q.pop()
            R.V.add(v)
            foreach (Kante e = (v,u) inzident zu v):
                R.E.add(e)
                if (!Discovered[u]):
                    Discovered[u] = true
                    Q.push(u)

        # Entferne alle künstlich hinzugefügten Kanten
        foreach (Kante e ∈ Added):
            R.E.remove(e)

        Result.add(R)

    return Result
```

Aufgabe 3.

(a) Bestimmen Sie für den nachfolgenden DAG alle topologischen Sortierungen. Es genügt entsprechend gereihte Listen von Knoten anzugeben.

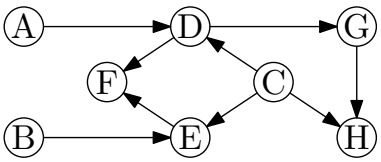
`E,B,A,D,(C|F)`

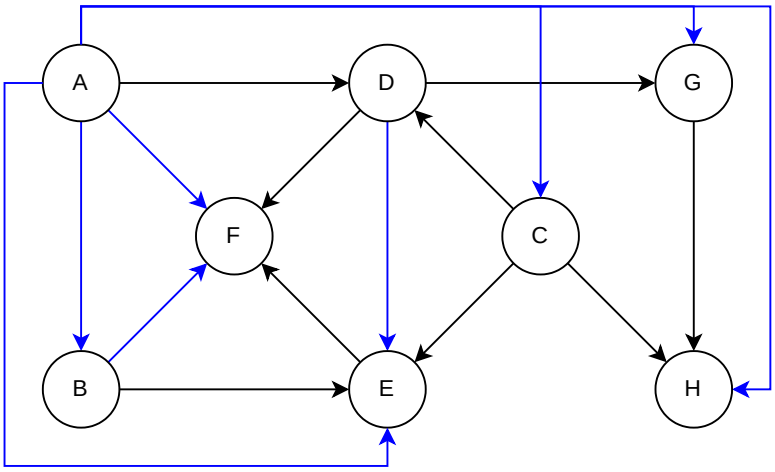


(b) Angenommen es wird beim DAG aus Unteraufgabe (a) eine Kante von C zu E hinzugefügt. Lässt sich für den resultierenden Graphen noch eine topologische Sortierung angeben? Falls ja, geben Sie diese an. Falls nein, begründen Sie kurz warum nicht.

`Nein, da dann der Graph nicht mehr azyklisch wäre`

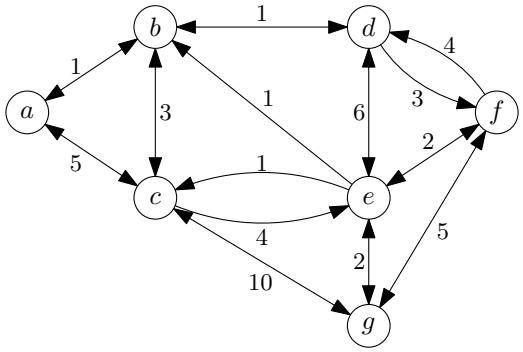
(c) A, B, C, D, E, F, G, H ist eine gültige topologische Sortierung in folgendem DAG. Fügen Sie acht weitere Kanten ein, sodass dies weiterhin eine gültige topologische Sortierung ist.





Aufgabe 4. Lösen Sie folgende Unteraufgaben zum Thema *„Kürzeste Pfade“*.

(a) Führen Sie den Algorithmus von Dijkstra auf dem nachfolgenden Graphen mit a als Startknoten aus. Geben Sie dabei am Ende jedes Durchlaufes der äußeren Schleife den Inhalt des Distanzarrays `d` an, und welcher Knoten neu als `Discovered` markiert wurde. Geben Sie außerdem den kürzesten Pfad von a nach g an.



Discovered	d(a)	d(b)	d(c)	d(d)	d(e)	d(f)	d(g)
a	0	1	5	∞	∞	∞	∞
a,b	0	1	4	2	∞	∞	∞
a,b,d	0	1	4	2	8	5	∞
a,b,c,d	0	1	4	2	5	5	14
a,b,c,d,e	0	1	4	2	5	5	7
a,b,c,d,e,f	0	1	4	2	5	5	7
a,b,c,d,e,f,g	0	1	4	2	5	5	7

(b) Angenommen, Sie sind nicht am kürzesten Pfad zwischen a und g interessiert, sondern an dem Pfad, welcher die wenigsten Knoten beinhaltet. Im Graph von Unteraufgabe (a) wäre dies der Pfad a, c, g mit 3 Knoten. Beschreiben Sie einen Algorithmus, welcher in einem gegebenen gerichteten Graphen G die minimale Anzahl an Knoten auf einem Pfad zwischen zwei gegebenen Knoten s und t berechnet. Ihr Algorithmus soll eine strikt bessere Laufzeit als $O(|E| + |V| \log |V|)$ aufweisen.

Eine ausreichend detaillierte Beschreibung des Algorithmus mit Begründung der Korrektheit und Laufzeit genügt. Es ist kein Pseudocode notwendig.

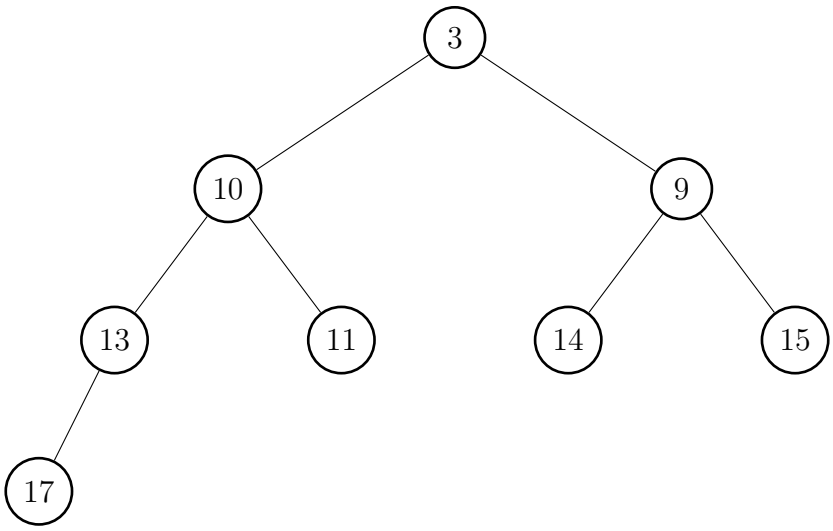
Algorithmus: Alle Gewichte auf 1 setzen und dann den A* Algorithmus ausführen.

Begründung der Korrektheit: Eigentlich trivial. Wenn alle Kanten gleich lang dauern ist der kürzeste Weg derjenige mit den wenigsten Kanten und damit auch wenigsten Knoten.

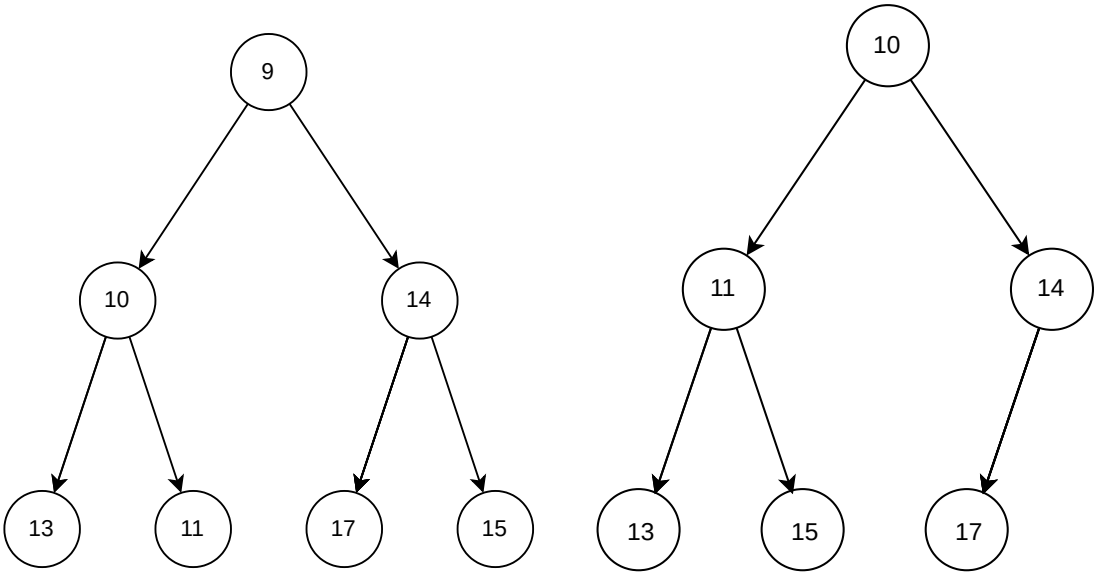
Begründung der Laufzeit:
- Alle Gewichte auf 1 setzten: O(|E|)
- A* Algorithmus: O(|E| log |V|)
- Summe: O(|E| log |V|)

```
Dijkstra(G,s):
Discovered[v] ← false für alle Knoten v ∈ V
d[s] ← 0
d[v] ← ∞ für alle anderen Knoten v ∈ V \ {s}
Q ← V
while Q ist nicht leer
    wähle u ∈ Q mit kleinstem Wert d[u]
    lösche u aus Q
    Discovered[u] ← true
    foreach Kante e = (u,v) ∈ E
        if !Discovered[v]
            if d[v] > d[u]+ℓ_e
                lösche v aus Q
                d[v] ← d[u]+ℓ_e
                füge v zu Q hinzu
```

Aufgabe 5. Gegeben ist der folgende **Min-Heap**:

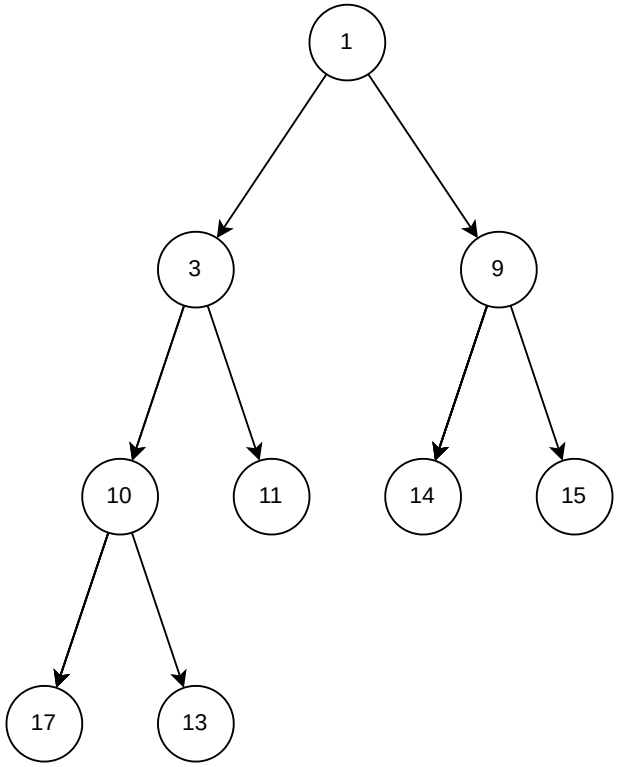


(a) Entnehmen Sie zwei Mal das kleinste Element. Stellen Sie nach jedem Löschvorgang den resultierenden Heap als Baum dar. Erklären Sie dabei, wie Sie **Heapify-down** korrekt anwenden.




```
Knoten 17 löschen und Knoten 3 auf 17 setzen.
Erste Iteration: 9 < 10 => 9 wird ausgewählt.
9 < 17 => Elemente vertauschen.
Zweite Iteration: 14 < 15 => 14 wird ausgewählt.
14 < 17 => Elemente vertauschen.
2*6 > n (n=7) => Done.

Knoten 15 löschen und Knoten 9 auf 15 setzen.
Erste Iteration: 10 < 14 => 10 wird ausgewählt.
10 < 15 => Elemente vertauschen.
Zweite Iteration: 11 < 13 => 13 wird ausgewählt.
13 < 15 => Elemente vertauschen.
2*4 > n (n=7) => Done.
```

(b) Betrachten Sie den ursprünglichen Heap, und fügen Sie das Element 1 ein. Stellen Sie nach dem Einfügevorgang den resultierenden Heap als Baum dar. Erklären Sie dabei, wie Sie **Heapify-up** korrekt anwenden.




```
1 wird an Stelle n+1 (=9) eingefügt.
Erste Iteration: i/2 = 4; H[4] = 13; 1 < 13 => Elemente vertauschen
Zweite Iteration: i/2 = 2; H[2] = 10; 1 < 10 => Elemente vertauschen
Dritte Iteration: i/2 = 1; H[1] = 3; 1 < 3 => Elemente vertauschen
i <= 1 => Abbruch
```

(c) Ein bekannter Sortieralgorithmus, der einen Heap als zentrale Datenstruktur verwendet, ist *Heapsort*. Der Algorithmus besteht aus folgenden zwei Phasen.

- Erst erstelle einen Min-Heap, der die Elemente des Eingabearrays enthält.
- Danach, solange der Heap nicht leer ist, entferne wiederholt das kleinste Element. Platziere die entfernten Elemente der Reihe nach im Ausgabearray.

Analysieren Sie die asymptotische Laufzeit dieses Sortieralgorithmus, und geben Sie eine möglichst geringe obere Schranke in O-Notation an.

```
Heap erstellen: O(n)
Kleinstes Element entfernen: O(log n)
Alle Elemente werden entfernt => Element entfernen wird n mal ausgeführt.
Laufzeit: O(n log n)
```

```
Heapify-up(H,i):
if i > 1
    j ← ⌊i/2⌋
    if H[i] < H[j]
        Vertausche die Array-Einträge H[i] und H[j]
        Heapify-up(H,j)
```

```
Heapify-down(H,i):
n ← length(H)-1
if 2·i > n
    return
elseif 2·i < n
    left ← 2·i,  right ← 2·i+1
    j ← Index des kleineren Wertes von H[left] und H[right]
else
    j ← 2·i
if H[j] < H[i]
    Vertausche die Arrayeinträge H[i] und H[j]
    Heapify-down(H,j)
```