Prof. Dr. Heiner Klocke

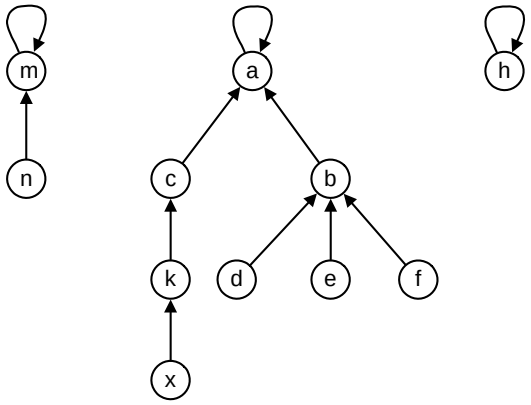
Themen:
- Schnelle Mengen-Operationen durch Union-Find-Strukturen
- Anwendungsbeispiel: Berechnung eines spannenden Baumes mit minimalen Kosten nach dem Algorithmus von Kruskal

Literatur: Kapitel 3 der Vorlesung

Aufgabe 1 (**ADT UNIONFIND**)

Entwickeln Sie eine Java-Klassenbibliothek für den ADT UNIONFIND aus Kapitel 3 der Vorlesung. Stellen Sie die Mengen, wie in dem Bild gezeigt, als Mehrwegebäume dar. Das Bild zeigt die Kollektion K der drei Mengen m, a und h:

K = {{m, n}, {a, c, b, k, d, e, f, x}, {h}}



a) Definieren Sie den ADT UNIONFIND und dokumentieren Sie die Strukturen des Baummodells.

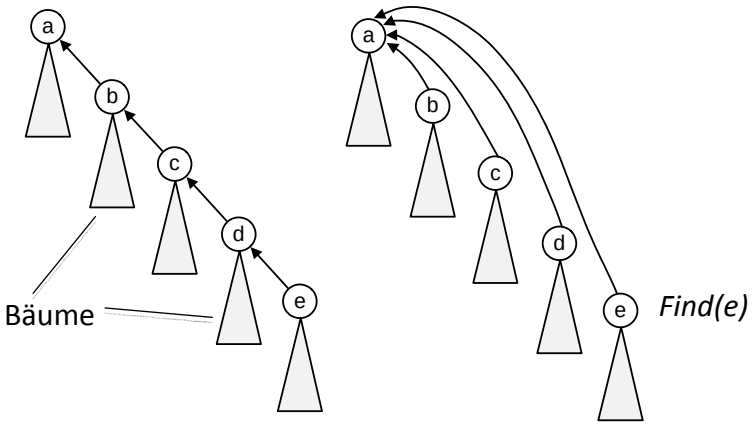
b) Implementieren Sie den ADT UNIONFIND in Java. Entwickeln Sie eine Klasse ***UnionFind***, mit der sich **Kollektionen** beliebiger Mengen bilden lassen, wie in dem obigen Beispiel dargestellt.

c) Entwickeln Sie eine **interaktive Testumgebung**, mit der Sie die Funktionen des ADT UNIONFIND interaktiv testen können. Geben Sie jeweils das Ergebnis einer ADT-Operation wie *Union* oder *Find* aus. Stellen Sie eine Kollektion K von Mengen sowohl in der **Mengenschreibweise**, z.B. { {m, n} {a, c, b, f, d, e, x, k} {h} } als auch wie oben gezeigt als **Wald von Mehrwegebäumen** dar.

d) Implementieren Sie für die Operationen *Union* und *Find* die **Heuristiken**

- **Vereinigung nach Größe** bei *Union(i, j)*. Bei der *Vereinigung nach Größe* zweier Bäume mit Wurzeln *i* und *j* wird die Wurzel des Baumes mit der kleineren Größe zum direkten weiteren Sohn des Baumes mit der größeren Größe. Kanonisches Element der Vereinigungsmenge wird die Wurzel des neuen Baumes.

- **Pfadverkürzung** bei *Find(x)*. Wie im folgenden Bild gezeigt, werden bei jedem Aufruf von *Find(x)* alle Bäume auf dem Pfad von x zum kanonischen Element direkt an die Wurzel gehängt.



Pseudocodes der Union-Find-Operationen (vgl. Folie 29 Kap 3) und der Heuristiken:

```
var p: array [element] of element;
procedure Make-Set(x : element);
begin
   p[x] := x;
end

procedure Union(e, f : element);
begin
   p[f] := e;
end
```

```
function Find(x : element) :
element;
Var y : element;
begin
   y := x;
   while p[y] ≠y do y := p[y];
   Find := y
end
```

Erste Heuristik: „Vereinigung nach Größe“ bei *union(e, f)*:

```
procedure Make-Set (x : element);
begin
   p[x] := x;
   Größe[x] := 1
end
```

```
procedure Union (e, f : element);
begin
If Größe[e]  <Größe[f]   then vertausche(e, f);
   { jetzt ist e das kanonische Element der größeren Menge}
   p[f] := e;
   Größe[e] := Größe[f] + Größe[e]
end
```

Zweite Heuristik: „Pfadverkürzung“ bei *find(e)*:

```
Find(e) {
 if  (parent(e) == e)  { return e;}
 parent(e) := find(parent[e]);
 return parent[e];
}
```

Aufgabe 2 (Anwendung – Kruskals Algorithmus)

Ein **bekanntes Problem** ist folgendes: Verschiedene Städte sind durch Straßen verbunden. Finde einen Straßenpfad, der alle Städte miteinander verbindet und dabei die kürzeste Gesamtstrecke ergibt.

In der Graphentheorie wird dieses Problem wie folgt formuliert:

> Gegeben ist ein Graph G = (V, E) mit Knotenmenge V (engl. vertex) und Kantenmenge E (engl. edge). Jeder Kante ist eine reelle Zahl c(e) als Kosten (engl. cost) zugeordnet. Der Graph G sei ungerichtet und zusammenhängend. Berechne einen spannenden Baum mit minimalen Kosten (Minimum Cost Spanning Tree (MCST)).

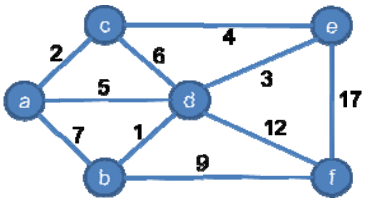


Bild 1: zusammenhängender, ungerichteter gewichteter Graph G

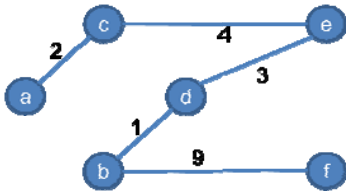


Bild 2: MCST zu G

Es verschiedene Algorithmen, die dieses Problem lösen. In dieser Aufgabe soll der spannende Baum mit dem **Algorithmus von Kruskal** berechnet werden (Vorlesung Kapitel 3, Folien 18-22). Der Algorithmus konstruiert Teilpfade des MCST. Die Teilpfade werden solange erweitert, bis alle Knoten verbunden sind. Die dazu verwenden Äquivalenzklassen (Teilpfade) sollen als Mengen durch UnionFind-Strukturen dargestellt werden. Weiterhin sollen die Heuristiken ***Vereinigung nach Größe*** und ***Pfadverkürzung*** eingesetzt werden! Die Implementierung soll sich an dem folgenden Pseudocode **MCST** (Minimum Cost Spanning Tree) orientieren, der die UnionFind-Operationen *Make-Set(x), Union(x,y)* und *Find(x)* benutzt dazu benutzt, die Äquivalenzklassen dynamisch zu erweitern.

```
Procedure MCST ((V, E) : Graph)
{ berechne zu einem zusammenhängenden, ungerichteten, gewichteten Graphen G = (V, E) einen
minimalen spannenden Baum T = (V, E') }
begin
    E' := 0;     K := 0;
    sortiere die Kanten aufsteigend nach den Gewichten in eine Liste Q
    for all v ∈ V do Make-Set(v);
    { jetzt besteht K aus allen Mengen {v}, v ∈ V }
    while K enthält mehr als eine Menge do
    begin
        (v, w) := min(Q); deletemin(Q);
        if  Find(v) ≠ Find(w) then
            begin
                Union(v0, w0), mit v0 = Find(v), w0 = Find(w);
                E' := E'∪ { (v, w) }
            end
    end
end
```

Bevor der Algorithmus mit der Konstruktion des MCST beginnt, müssen die Kanten von G aufsteigend nach den Gewichten in eine Liste Q sortiert werden. Die kleinste Kante steht jeweils am Anfang der Liste und wird durch *min(Q)* referenziert und durch *deletemin(Q)* entfernt.

Der Graph kann als Adjazenzmatrix repräsentiert werden:

	a	**b**	**c**	**d**	**e**	**f**
a	0	7	2	5	∞	∞
b	7	0	∞	1	∞	∞
c	2	∞	0	6	∞	∞
d	5	1	6	0	3	12
e	∞	∞	4	3	0	17
f	∞	9	∞	12	17	0

- Erzeugen Sie den Graph G = (V, E) mit einem Zufallsgenerator: ca. 100 Knoten V und ca. 2000-4000 gewichtete Kanten E
- Sortieren Sie die Kanten E aufsteigend nach Gewichten
- Erzeugen Sie zu jedem Knoten $k \in V$ mit der Operation *MakeSet(k)* eine Äquivalenzklasse {k}.
- Stellen Sie die während der Ausführung des Algorithmus dar, wie sich die Teilpfade (Mengen der Kanten) durch die **UnionFind-Operationen** *Union* und *Find* verändern.
- Wichtig ist die Wahl einer geeigneten grafischen Darstellung, in der die dynamischen Veränderungen der **Höhen** aller UnionFind-Bäume sichtbar werden. ($\rightarrow$ Bild 3)
- Verwenden Sie einen interaktiven Schalter, mit dem die beiden Heuristiken „Vereinigung nach Größe“ und „Pfadverkürzung“ ein- und ausgeschaltet werden können.
- Zählen Sie die **Anzahl f der ausgeführten *Find*- Operationen**, die benötigt werden, um den MCST für eine Menge von **n Knoten** zu berechnen.
- Vergleichen Sie die gezählte **Anzahl der Aufsteige-Schritte** mit der in dem folgenden Satz vorhergesagten Anzahl $\Theta(m * \alpha(m, n))$. m = f + n

Was sagt die **Theorie**, wenn **m = f + n** Operationen ausgeführt werden, wobei **f** = Anzahl der *Find*-Operationen und **n** = Anzahl der *Make-Set*-Operationen?

Satz (Heuristiken: Vereinigung nach Größe und Pfadverkürzung)

Durch die Verwendung der Strategien *Vereinigung nach Größe* und *Pfadverkürzung* benötigen **m** Find-Operation über **n** Elementen $\Theta(m * \alpha(m, n))$ Schritte. Mit $\alpha(m, n)$ wird die Inverse der Ackermann-Funktion bezeichnet.

Ackermann-Funktion

$A(1, j) = 2^j$ für $j \geq 1$
$A(i, 1) = A(i - 1, 2)$ für $i \geq 2$
$A(i, j) = A(i - 1, A(i, j - 1))$ für $i, j \geq 2$

Inverse Ackermann-Funktion

$$\alpha(m,n) = \min\left\{ i \geq 1 \middle| A(i, \lfloor m/n \rfloor) > \log n \right\}$$

Die dynamischen Veränderungen der Höhen der UnionFind-Bäume durch die Operationen *Find* und *Union* können in einem Diagramm dargestellt werden. Durch die Anwendung der Heuristiken werden sich die Baumhöhen stark verkürzen, was in dem Diagramm beobachtet werden kann. Geben Sie daher eine solche Darstellungsform auf dem Bildschirm aus.

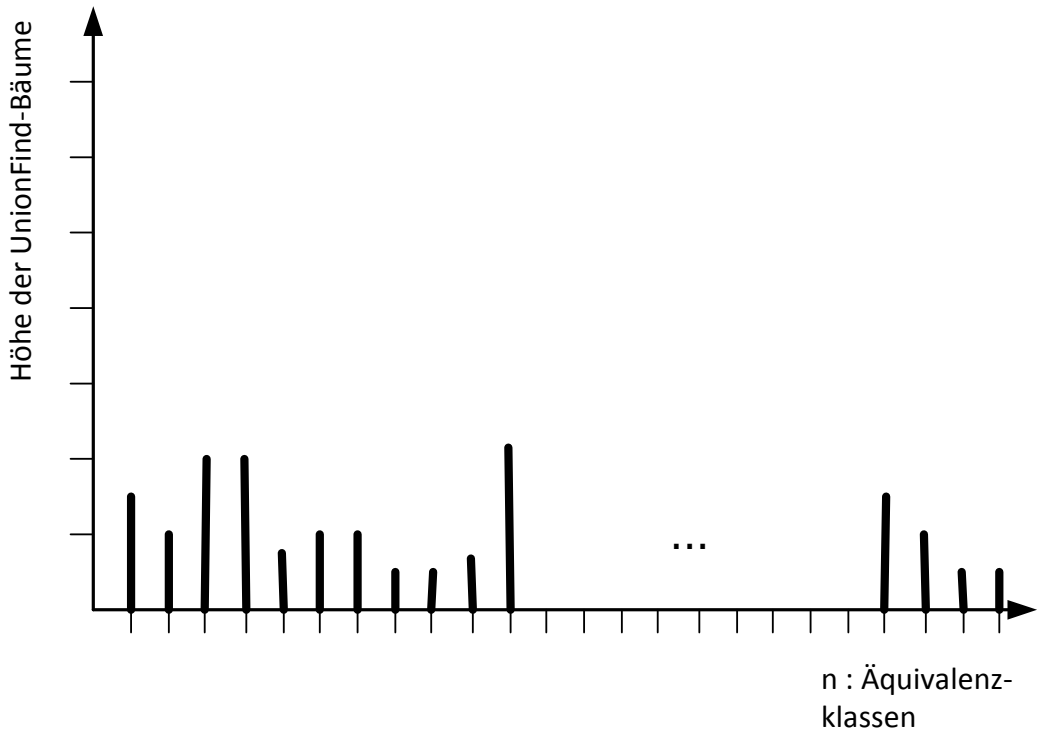


Bild 3: Darstellung der dynamischen Veränderungen der Baumhöhen durch Anwendung der Operationen *Find* und *Union*. Durch Union verändert sich ebenfalls die Anzahl der Äquivalenzklassen (Mengen).

Beachten Sie, dass sich bei Kruskals Algorithmus die Anzahl der anfangs n Äquivalenzklassen verringert. Am Ende gibt es nur noch eine einzige Klasse mit allen Knoten, die durch einen Pfad mit minimalen Kosten verbunden sind. Wir erwarten nach der Theorie, dass der Baum, der die Klasse mit allen Knoten repräsentiert, eine sehr geringe Höhe hat.

Eigene Ideen zur Darstellung der UnionFind-Bäume und ihrer Strukturänderungen sind erwünscht!

Bei Fragen bitte melden!