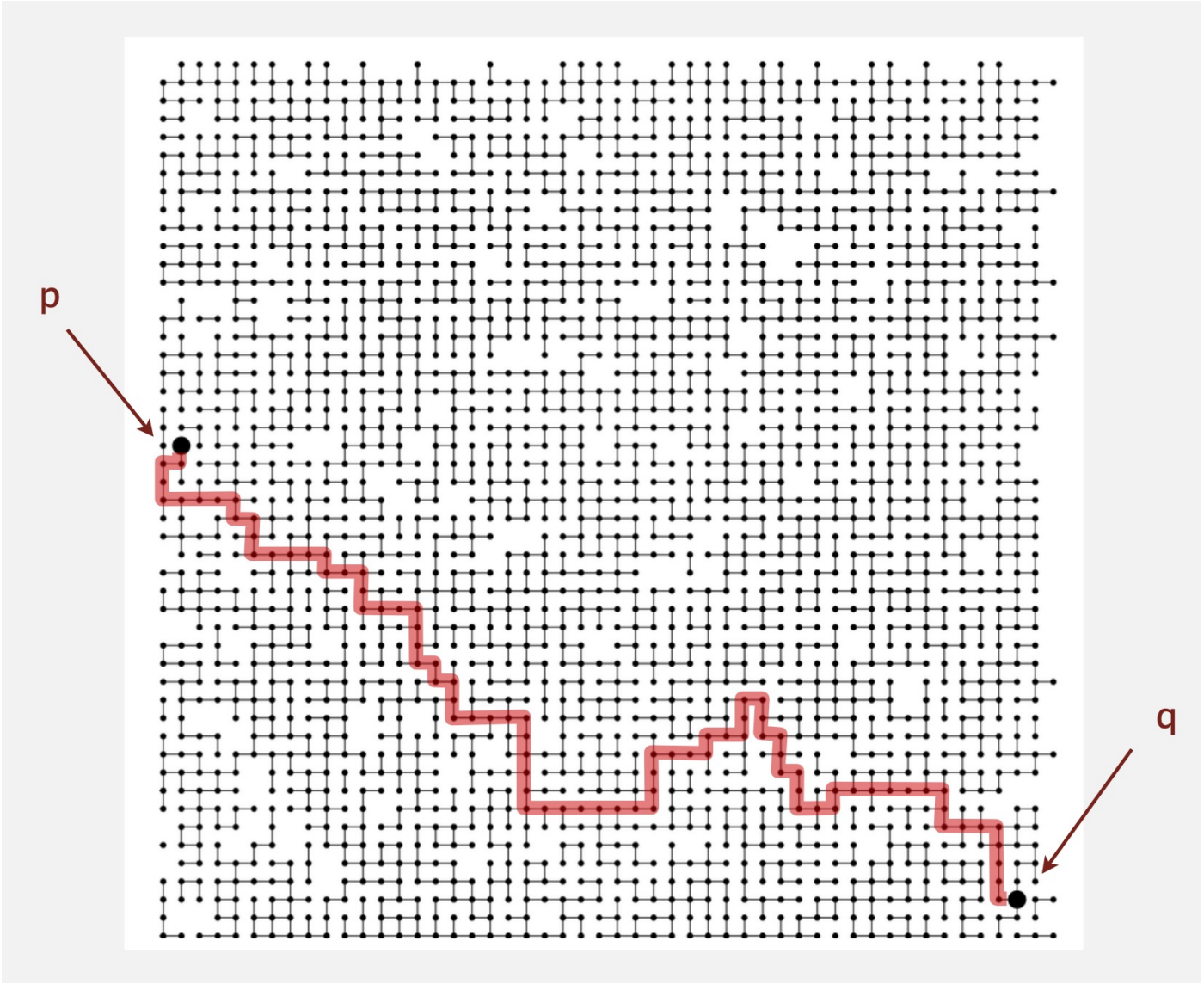
14. Union-Find-Struktur

- Motivation
- Problemstellung Partitionierung
- Partitionierung als Wald
  mit Elternfeld als Datenstruktur
- Find-Algorithmus
- Union-By-Height und Union-By-Size
- Pfad-Kompression und Union-By-Rank
- Analyse
- Implementierung einer generischen Union-Find-Struktur
- Anwendungen

Motivation - Konnektivitätsproblem

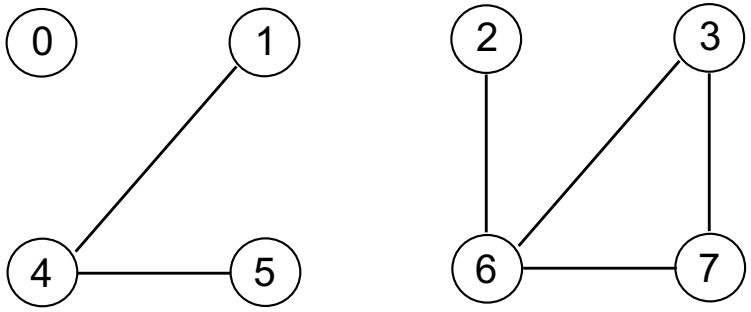


- Sind Knoten p und Knoten q durch einen Weg verbunden?
- Frage soll auch effizient beantwortet werden für andere Knotenpaare p und q!
- Außerdem dürfen neue Kanten zum Graph hinzugefügt werden.

Ungerichteter Graph aus https://algs4.cs.princeton.edu/lectures/keynote/15UnionFind.pdf

Lösungsansatz für Konnektivitätsproblem

- Zerlege Knotenmenge V in disjunkte Teilmengen (Partitionierung), so dass jede Teilmenge alle Knoten enthält, die miteinander über einen Weg verbunden sind.



$V = \{0\} \cup \{1,4,5\} \cup \{2,3,6,7\}$

Beachte: Teilmengen sind disjunkt!

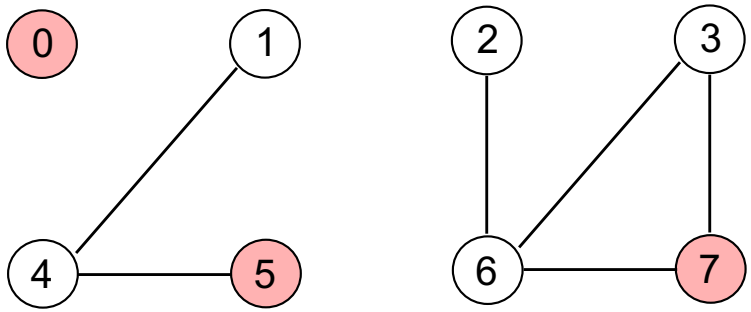
- Knoten p und q sind genau dann durch einen Weg verbunden, falls p und q in derselben Teilmenge enthalten sind.

Problemstellung

- Verwalte Partionierung einer Menge G.
  D.h. Zerlegung der Menge G in disjunkte Teilmengen.
- Operation find(x) liefert irgendein Element (Repräsentant)
  aus der Teilmenge zurück, zu der x gehört.
  Außerdem: find(x) = find(y) gdw. x und y zur selben Teilmenge gehören.
- Operation union(x,y) vereinigt die Teilmenge, zu der x gehört, mit der Teilmenge, zu der y gehört.
- Die Partitionierung ist daher dynamisch!
- Die Datenstruktur, die die Operationen union und find anbietet wird auch Union-Find-Struktur genannt.

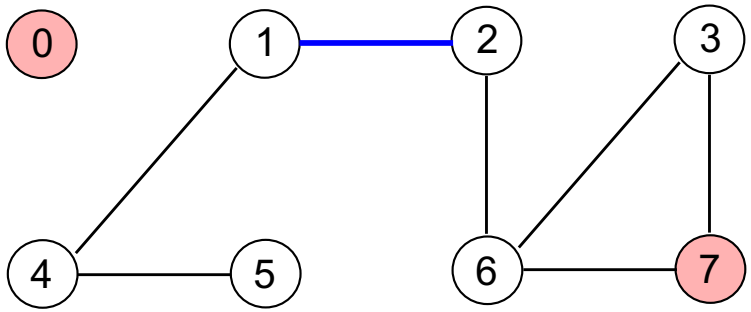
Beispiel zu Konnektivitätsproblem

- find(2) = find(3) = 7
  Also: 2 und 3 gehören zur selben Teilmenge.
  Knoten 7 ist als Repräsentant seiner Teilmenge hier willkürlich gewählt.



V = {0} ∪ {1,4,5} ∪ {2,3,6,7}
0, 5 und 7 sind Repräsentanten

- Füge eine Kante (1,2) ein und führe daher union(1,2) aus:



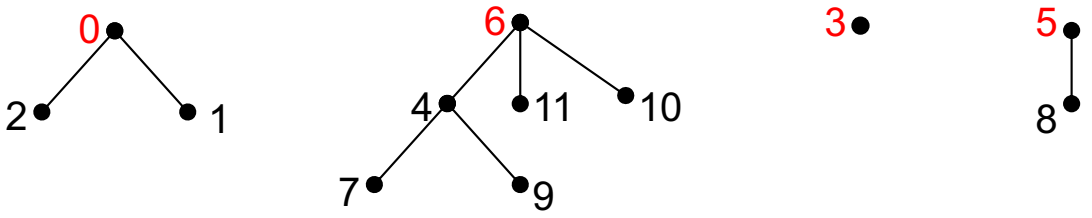
V = {0} ∪ {1,4,5,2,3,6,7}
7 bleibt Repräsentant.

Partitionierung als Wald

- Speichere Teilmengen einer Partitionierung als Bäume.
- Die Wurzel ist der Repräsentant der jeweiligen Teilmenge.
- Alle Teilmengen einer Partitionierung ergeben daher einen Wald (Menge von Bäumen)
- Beispiel:

$G = \{0, 1, 2\} \cup \{4, 6, 7, 9, 10, 11\} \cup \{3\} \cup \{5, 8\}$ Partitionierung.

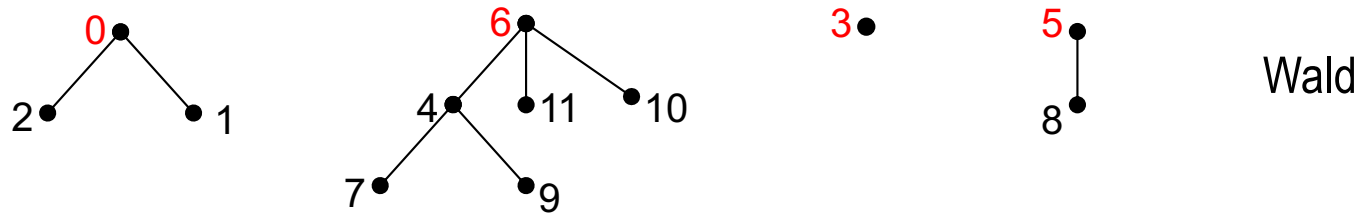
Repräsentant der jeweiligen Teilmenge.



Wald

Wald mit Elternfeld als Datenstruktur

- Annahme: Verwaltung einer Partitionierung von G = {0, 1, 2, 3, ..., n-1}.
  Elemente können dann geschickterweise als Feldindex verwendet werden.
- Wald als Elternfeld p, wobei p[x] der Elternknoten von x ist.
- Falls x eine Wurzel ist, dann enthält p[x] einen negativen Wert.
  Zum Beispiel p[x] = -1.



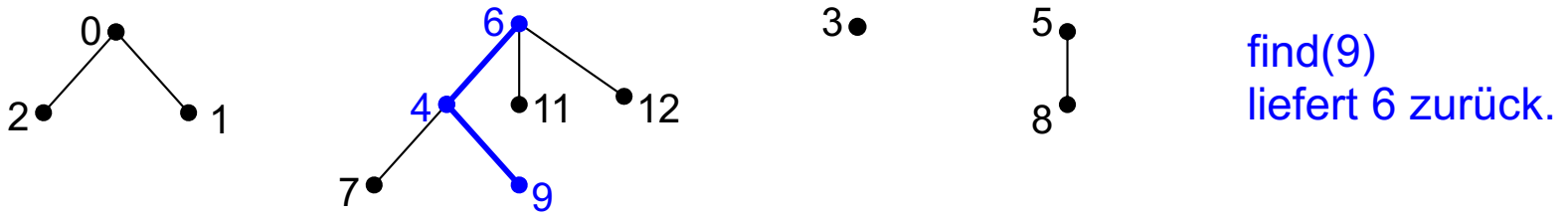
x	0	1	2	3	4	5	6	7	8	9	10	11
p[x]	-1	0	0	-1	6	-1	-1	4	5	4	6	6

Elternfeld

Hier noch -1. Später anderer negativer Wert

Find-Algorithmus

```
int find(int x) {
      while (p[x] >= 0)     // x ist keine Wurzel
            x = p[x];
      return x;
}
```



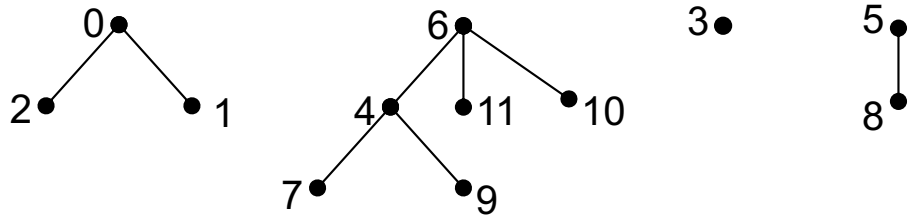
find(9)
liefert 6 zurück.

x	0	1	2	3	4	5	6	7	8	9	10	11
p[x]	-1	0	0	-1	6	-1	-1	4	5	4	6	6

- Beachte: x und y gehören zur selben Menge gdw. find(x) = find(y)

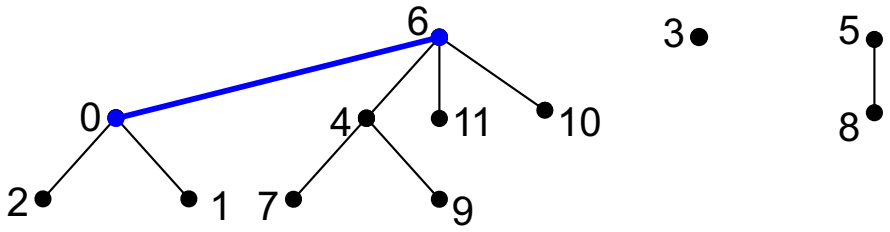
Union-Algorithmus - Idee

- Werden zwei Mengen vereinigt, dann wird der kleinere Baum als Kind der Wurzel des größeren Baums eingehängt.
- Beispiel:



x	0	1	2	3	4	5	6	7	8	9	10	11
p[x]	-1	0	0	-1	6	-1	-1	4	5	4	6	6

union(2,4) vereinigt die Menge {0, 1, 2} und die Menge {4, 6, 7, 9, 10, 11}.



x	0	1	2	3	4	5	6	7	8	9	10	11
p[x]	6	0	0	-1	6	-1	-1	4	5	4	6	6

- Varianten: Wähle als Größe die Höhe bzw. die Knotenanzahl.
  → Union-By-Height bzw. Union-By-Size

Union-By-Height

- Es wird der Baum mit der kleineren Höhe an die Wurzel des Baums mit der größeren Höhe gehängt.
- Die Höhe h für ein Baum lässt sich bei p[w], wobei w die Wurzel ist, als negative Zahl abspeichern: p[w] = -1-h

```
void unionByHeight (int x1, int x2) {

        w1 = find(x1);
        w2 = find(x2);
        if (w1 == w2) return;

        if ( -p[w1] < -p[w2] )   // Höhe von w1 < Höhe von w2
                p[w1] = w2;
        else {
                if (p[w1] == p[w2] ) // w1 und w2 sind gleich hoch
                        p[w1]--;   // Höhe von w1 erhöht sich um 1
                p[w2] = w1;
        }
}
```

Union-By-Size

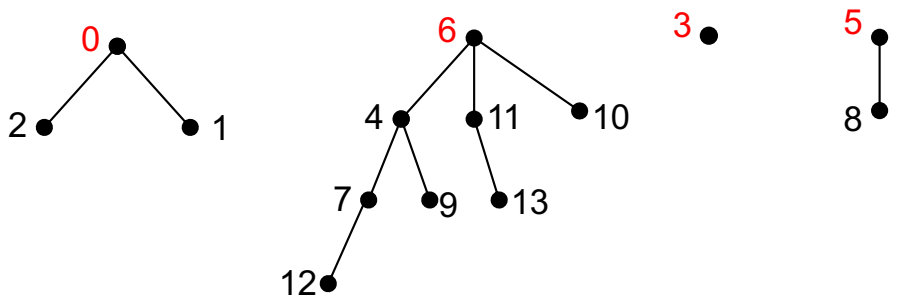
- Es wird der Baum mit geringerer Anzahl Knoten an die Wurzel des Baums mit größerer Anzahl Knoten gehängt.
- Die Knotenanzahl n für ein Baum lässt sich bei p[w], wobei w die Wurzel ist, als negative Zahl abspeichern: p[w] = -n

```
void unionBySize (int x1, int x2) {

        w1 = find(x1);
        w2 = find(x2);
        if (w1 == w2) return;

        if (-p[w1] < -p[w2] ) {   // Größe von w1 < Größe von w2
                p[w2] = p[w2] + p[w1];   // Größe von w2 wächst um Größe von w1
                p[w1] = w2;
        } else {
                p[w1] = p[w1] + p[w2];   // Größe von w1 wächst um Größe von w2
                p[w2] = w1;
        }
}
```

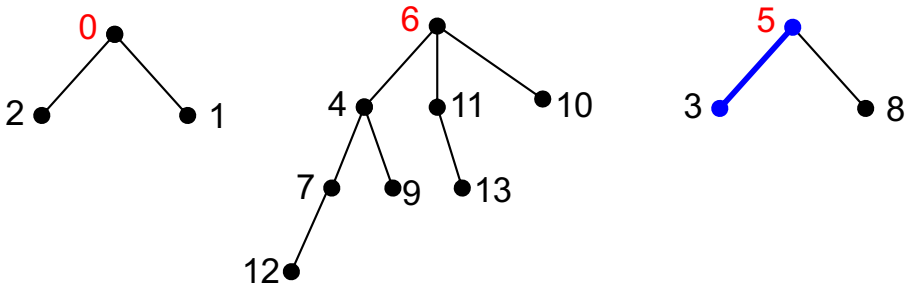
Beispiel für Union-By-Height (1)



e	0	1	2	3	4	5	6	7	8	9	10	11	12	13
p[e]	-2	0	0	-1	6	-2	-4	4	5	4	6	6	7	11

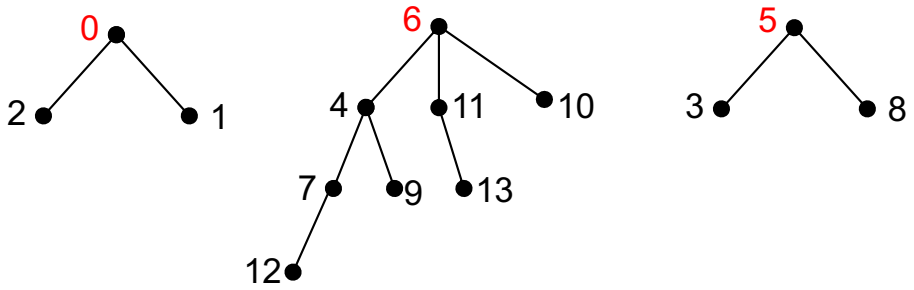
Wurzeln

Höheninformation

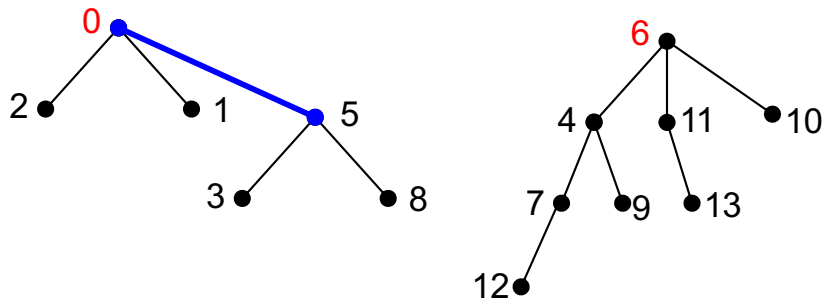


e	0	1	2	3	4	5	6	7	8	9	10	11	12	13
p[e]	-2	0	0	5	6	-2	-4	4	5	4	6	6	7	11

Beispiel für Union-By-Height (2)

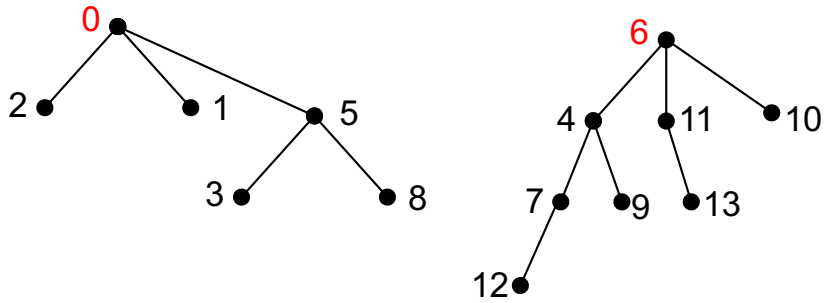


e	0	1	2	3	4	5	6	7	8	9	10	11	12	13
p[e]	-2	0	0	5	6	-2	-4	4	5	4	6	6	7	11

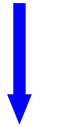


e	0	1	2	3	4	5	6	7	8	9	10	11	12	13
p[e]	-3	0	0	5	6	0	-4	4	5	4	6	6	7	11

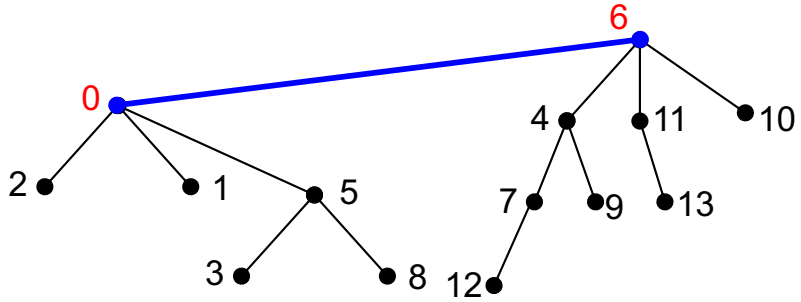
Beispiel für Union-By-Height (3)



e	0	1	2	3	4	5	6	7	8	9	10	11	12	13
p[e]	-3	0	0	5	6	0	-4	4	5	4	6	6	7	11



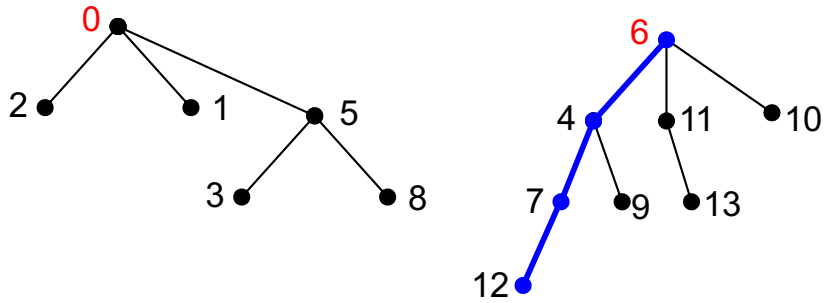
unionByHeight(3,11)



e	0	1	2	3	4	5	6	7	8	9	10	11	12	13
p[e]	6	0	0	5	6	0	-4	4	5	4	6	6	7	11

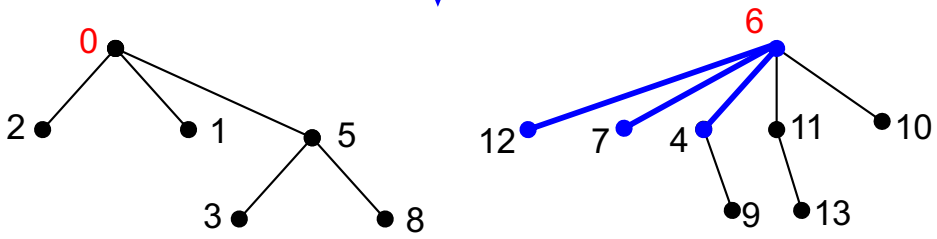
Find mit Pfadkompression (1)

- Idee: find(x) durchläuft Pfad p vom Knoten x bis zur Wurzel.
  Dabei werden alle Knoten (außer der Wurzel) des Pfads p direkt auf die Wurzel verlinkt.
- Der Pfad wird dadurch komprimiert.
- Da die Komplexität von find im Wesentlichen von der Baumhöhe bestimmt wird,
  hat die Pfadkompression über die Zeit gemittelt einen positiven Effekt auf die Laufzeit.



e	0	1	2	3	4	5	6	7	8	9	10	11	12	13
p[e]	-3	0	0	5	6	0	-4	4	5	4	6	6	7	11

findWithCompression(12)

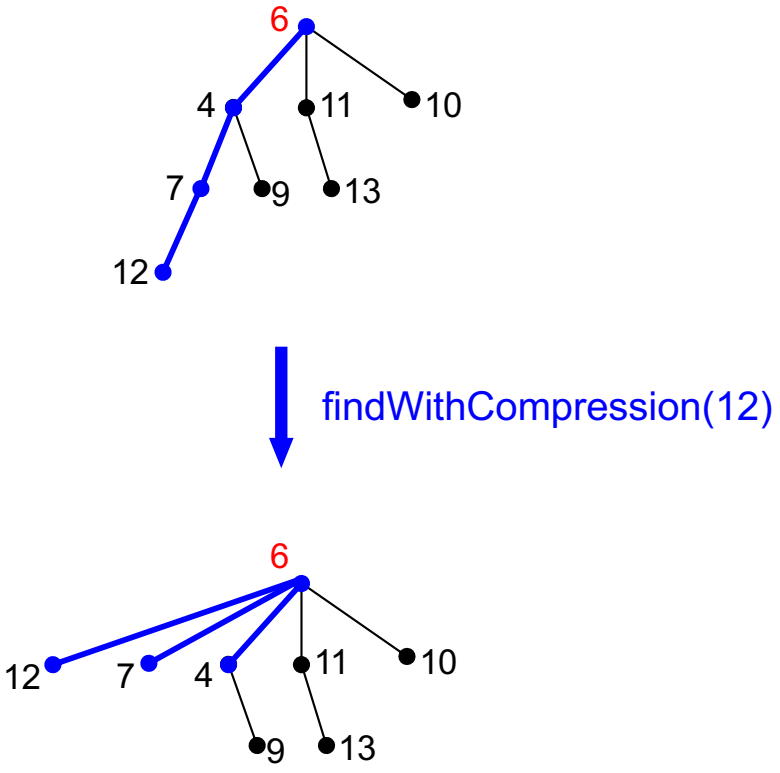


e	0	1	2	3	4	5	6	7	8	9	10	11	12	13
p[e]	-3	0	0	5	6	0	-4	6	5	4	6	6	6	11

Find mit Pfadkompression (2)

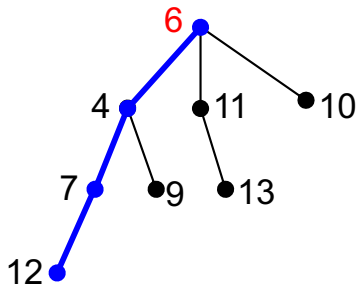
- Pfadkompression lässt sich am einfachsten rekursiv implementieren!

```
int findWithCompression(int x) {
    if (p[x] < 0)   // x ist Wurzel
        return x;
    else {
        p[x] = findWithCompression(p[x]);
        return p[x];
    }
}
```



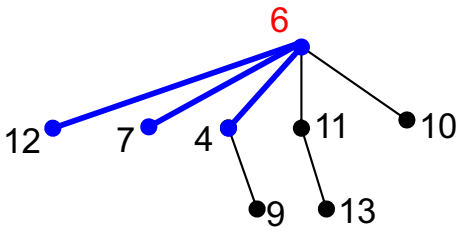
Union-By-Rank

- Wird find mit Pfad-Kompression verwendet, dann stimmt im allgemeinen die Höhenangabe bei der Wurzel nicht mehr.
- Union-by-Height kann dennoch unverändert verwendet werden. Das Union-Verfahren wird dann aber Union-By-Rank genannt. Die „Höheninformationen“ werden dabei auch Rank-Werte genannt.
- Union-by-Size ist dagegen kompatibel mit find mit Pfad-Kompression.



p[6] = -4. Höhe = 3.

findWithCompression(12)



Es bleibt p[6] = -4.
Jedoch Höhe = 2.

Analyse von Union-Find-Strukturen

1. unionByHeight und find mit n-Elementen haben eine Laufzeit von T(n) = O(log n).

   Es lässt sich induktiv zeigen, dass in der Union-Find-Struktur
   die Höhe jedes Baums höchstens log(n) ist. (siehe [Sedgewick, Algorithms])

2. Analog lässt sich zeigen, dass unionBySize und find eine Laufzeit von T(n) = O(log n) haben.

3. Eine sehr aufwendige Analyse zeigt, dass eine Folge von m unionByRank- und findWithCompression-Operationen eine Laufzeit von O(m log*(n)) haben.

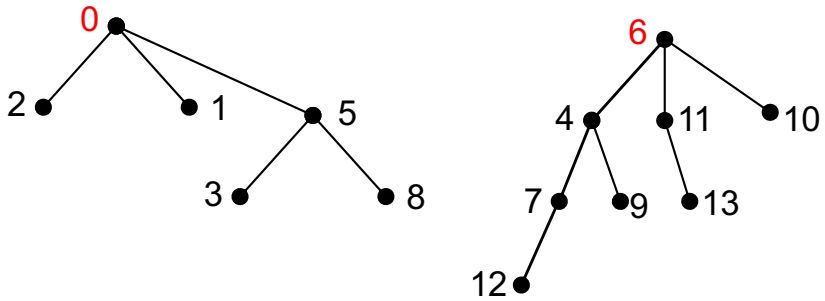
   Dabei ist log*(n) die Inverse der einstelligen Ackermann-Funktion.
   log*(n) gibt an wie oft n logarithmiert werden muss, bis 1 erreicht wird.
   Z.B. ist log*(65536) = 4, da log(log(log(log(65536)))) = 1. Und $\log^*(2^{65536}) = 5$.

   log*(n) wächst so langsam, dass praktisch O(m log*(n)) ≈ O(m) angenommen werden kann. D.h. im Durchschnitt haben unionByRank und findWithCompression praktisch eine Laufzeit von O(1).

   Siehe [Weiss, Data Analysis and Algorithm Analysis in Java].

Implementierung bei beliebigem Elementtyp

- Ein Elternfeld als Datenstruktur für eine Partitionierung setzt voraus, dass die Elemente als Indizes verwendet werden können. Also Menge der Elemente = {0, 1, 2, ..., n-1}.
- Sind die Elemente von einem beliebigen Typ T (z.B. Point, String, etc.), dann kann für das Elternfeld ersatzweise eine Map verwendet werden.
- Die Höheninformation (bzw. Anzahl Knoten bzw. Rank) wird in einer separaten Map verwaltet.



e	0	1	2	3	4	5	6	7	8	9	10	11	12	13
p[e]	-3	0	0	5	6	0	-4	4	5	4	6	6	7	11

Elternfeld p mit Verweisen auf Elternknoten bzw. Höheninformationen als negative Werte

```
Map<T,T> p = new HashMap<>();
Map<T,Integer> height = new HashMap<>();
```

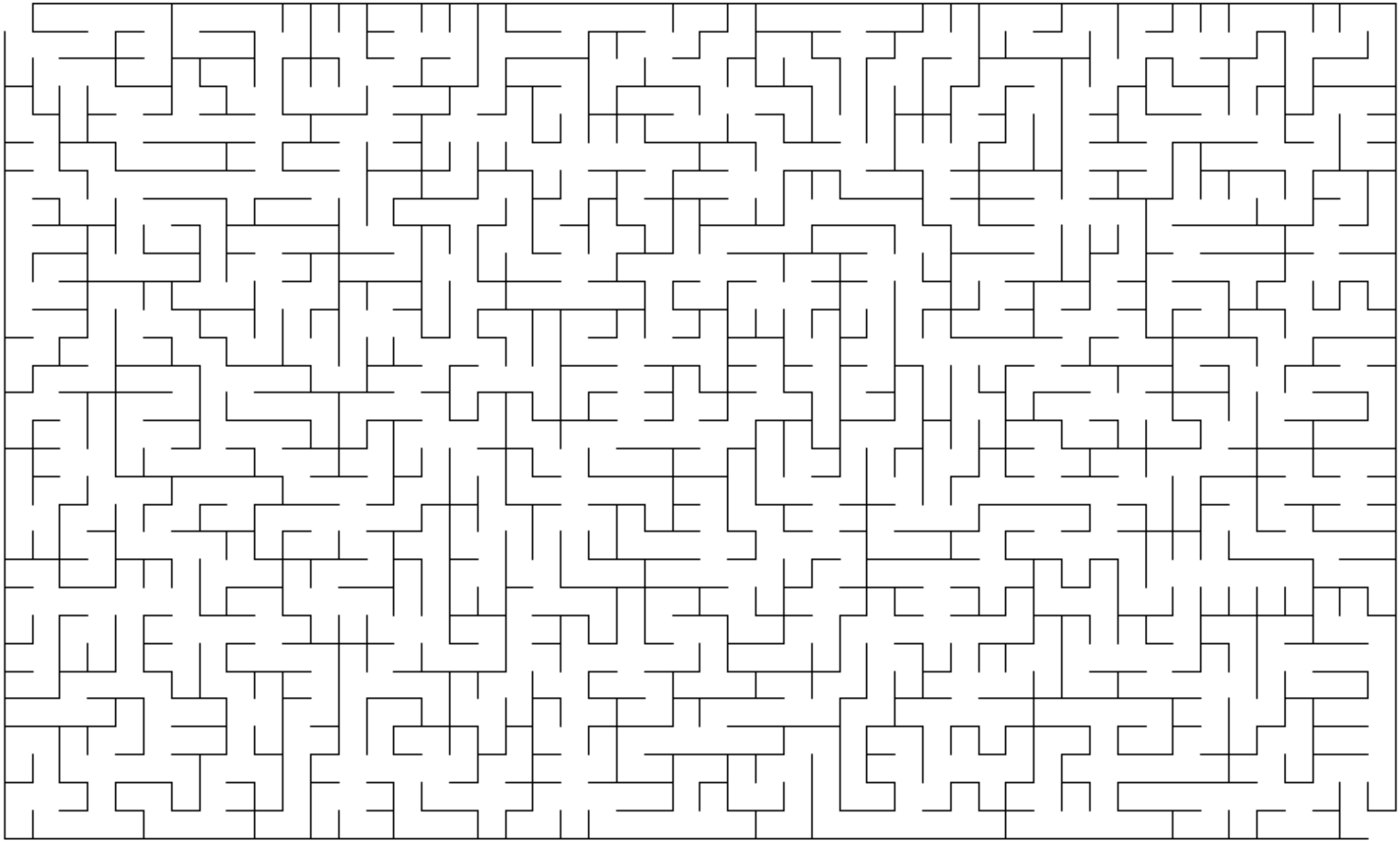
Map p, die für jeden Knoten den Elternknoten speichert. Wurzeln sind in p nicht enthalten!

Map height, die für jede Wurzel die Höheninformationen speichert.

14. Union-Find-Struktur

- Motivation
- Problemstellung Partitionierung
- Partitionierung als Wald
  mit Elternfeld als Datenstruktur
- Find-Algorithmus
- Union-By-Height und Union-By-Size
- Pfad-Kompression und Union-By-Rank
- Analyse
- Implementierung einer generischen Union-Find-Struktur
- Anwendungen

Generierung von Labyrinthen

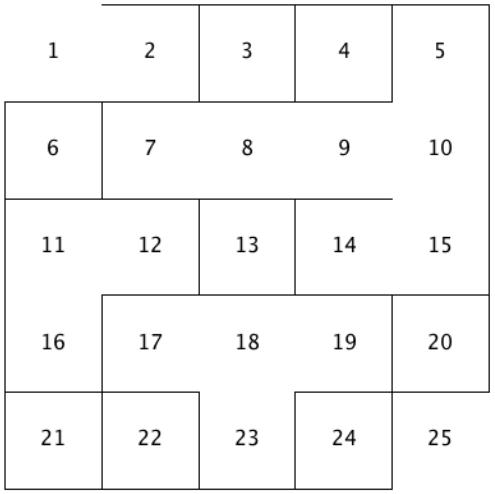


Generierung von Labyrinthen mit einer Union-Find-Struktur

- Idee:

  Union-Find-Struktur um Erreichbarkeit von Zellen in einem Labyrinth zu speichern: zwei Zellen des Labyrinths sind genau dann in der gleichen Teilmenge, falls sie gegenseitig erreichbar sind.

- Labyrinth

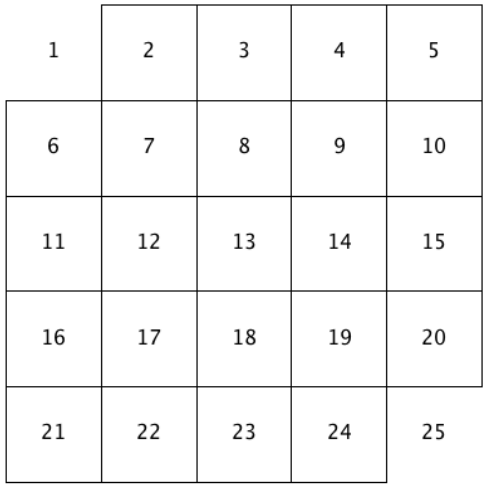


- Union-Find-Struktur

  { {1, 2}, {3}, {4}, {5, 7, 8, 9, 10, 14, 15}, {6}, {11, 12, 16}, {13}, {17, 18, 19, 23}, {20}, {21}, {22}, {24}, {25} }

Algorithmus (1) - Initialisierung

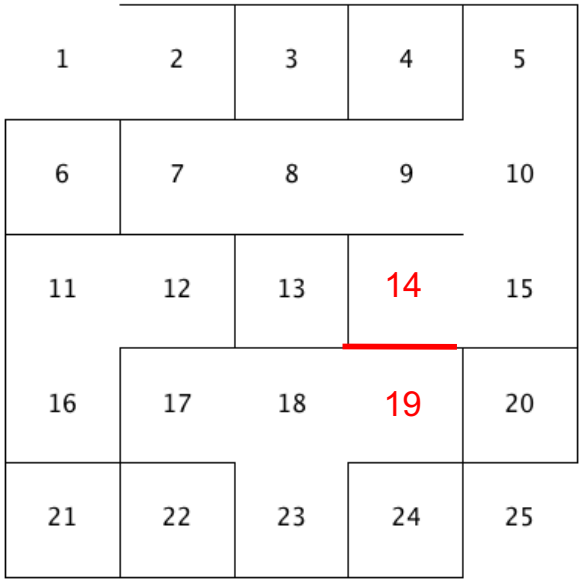
Initialisiere Labyrinth mit n*n Einzelzellen:



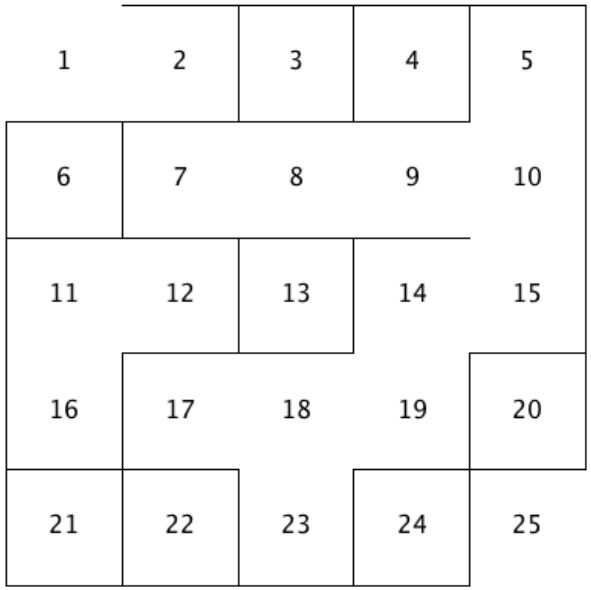
Union-Find-Struktur = { {1}, {2}, {3}, {4}, {5}, ... {25} }

Algorithmus (2): Wände beseitigen, so dass Erreichbarkeit erhöht wird

while (Anzahl Mengen in Union-Find-Struktur != 1) {
    wähle eine Wand w (keine Außenwand) zufällig aus;
    beseitige Wand w, falls dadurch
    die beiden benachbarten Zellen erreichbar werden;
}



{ {1, 2}, {3}, {4},
{5, 7, 8, 9, 10, 14, 15}, {6}, {11, 12, 16}, {13},
{17, 18, 19, 23}, {20}, {21}, {22}, {24}, {25} }

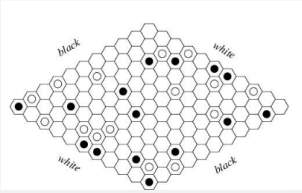


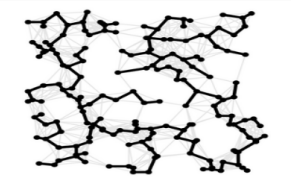
{ {1, 2}, {3}, {4},
{5, 7, 8, 9, 10, 14, 15, 17, 18, 19, 23},
{6}, {11, 12, 16}, {13}, {20}, {21}, {22}, {24}, {25} }

Weitere Anwendungen

Union-find applications

- Percolation.
- Games (Go, Hex).
- ✓ Dynamic connectivity.
- Least common ancestor.
- Equivalence of finite state automata.
- Hoshen-Kopelman algorithm in physics.
- Hinley-Milner polymorphic type inference.
- Kruskal's minimum spanning tree algorithm.
- Compiling equivalence statements in Fortran.
- Morphological attribute openings and closings.
- Matlab's `bwlabel()` function in image processing.






https://algs4.cs.princeton.edu/lectures/keynote/15UnionFind.pdf