Union-Find

FIND(x): O(1)
UNION(A,B,C): O(n)

```
for i=1 to n do
        name[i] <- i
od
```

```
Find: return name[x]
```

```
Union:
for i=1 to n do
        if name[i]==A OR name[i]==B
                then name[i] <- C
        fi
od
```

Relabel the smaller half:

FIND(x): O(1)
UNION(A,B): O(log(n))

```
Union:
if size[A] ≤ size[B]
then
        forall i in L[A] do
                name[i] <- B
        od
        size[B] += size[A]
        L[B] <- L[B] concatenate L[A]
else
        forall i in L[B] do
                name[i] <- A
        od
        size[A] += size[B]
        L[A] <- L[A] concatenate L[B]
fi
```

Weighted Union Rule: Hierbei gilt das Lemma, dass stets $size(x) \geq 2^{hoehe(x)}$ für alle Knoten gilt $\rightarrow$ $hoehe(x) = log(n)$

Find: O(log(n))
Union: O(1)

```
Init:
for 1=1 to n do
        vater[i] = 0
        name[i] = i
        wurzel[i] = i
        size[i] = 1
od
```

```
Find:
while vater[x] != null do
        x = vater[x]
od
return name[x]
```

```
Find mit Pfadkompression (immernoch O(log(n))):
r = x
while vater[r] != null do
        r = vater[r]
od
while x != r do
        y = vater[x]
        vater[x] = r
        x = y
od
return name[r]
```

```
Union:
r1 = wurzel[A]
r2 = wurzel[B]
if size[r1] <= size[r2] then
        vater[r1] = r2
        name[r2] = C
        wurzel[C] = r2
        size[r2]  += size[r1]
else
        vater[r2] = r1
        name[r1] = C
        wurzel[C] = r1
        size[r1]  += size[r2]
fi
```

Split-Find-Problem

Verwalte Liste mit Namen der jeweiligen Elemente.
Verwalte Liste mit Anfang und Ende jedes Intervalls.
Verwalte einen Zähler der den aktuell höchsten Namen speichert.
Find gibt name[x] zurück.
Split benennt die kleine Hälfte des betroffenen Intervalls um. Dafür verwendet es den Counter +1 als Namen.
Find offenstichich O(1). Split O(log(n))

```
public class Split_Find {

static int[] name;
static int count = 0;

static void init(int len) {
        name = new int[len];
        for(int i = 0; i<len; i++) {
                name[i] = 0;
        }
        ArrayToString();
}
static int find(int x){
        return name[x];
}
static void split(int x){
        Split_Find.count ++;
        int set = find(x);

        for(int i = 0; i < x; i++){
```

```
                if(name[i] == set)
                        name[i] = Split_Find.count;
        }
        ArrayToString();
}
```

Wörterbücher

Randomized Searchtree
Knotenorientierter Suchbaum für Paare $(x_i, prio(x_i))$. Maximum Heap bezüglich der Prioritäten. Die Prioritäten auf einem Pfad von der Wurzel zum Blatt sind monoton fallend.
Initialisierung: Weise jedem x eine zufällige Priorität zu. Füge die Schlüssel in der Reihenfolge absteigender Prioritäten normal in den Baum ein.
Lookup(x): Suche im binären Suchbaum O(Höhe(T))
Insert(x):Bestimme zufällige Priorität. Füge Knoten gemäß Schlüssel ein. Rotiere den Knoten nach oben, bis prio(vater(v)) $\geq$ prio(v) bzw v=Wurzel
Delete(x): Rotiere v nach unten, bis Blatt. Lösche dann. O(log(n))

```
while v kein Blatt do
        if prio(leftChild) > prio(rightChild) AND rightChild == null
                rotate_right(v)
        else
                rotat_left(v)
        fi
od
```

Hashing

h:Hashfunktion, **Hashing mit Verkettung**
Speichere für jedes Ergebnis der Hashfunktion eine Liste.
Lookup(x): Durchsuche T[h(x)] linear. wc:O(n), erw:O($1 + \frac{n}{m}$)
Insert(x): Füge an erste freie Stelle von T[h(x)] ein
Delete(x): Entferne aus T[h(x)]

Wenn der Belegungsfaktor n/m > 4, verdopple Tafelgöße
Hashing mit offener Adressierung
Folge von Hashfunktionen $h_i(x)$. Falls $h_1(x)$ belegt versuche $h_2(x)$ usw.
Lookup/Delete: Teste bis $T[h_i(x)] == x$.
Achtung: Verwalte Statusarray mit empty/full/deleted Einträgen, damit bekannt ist, ob lookup abbrechen soll.
Perfektes Hashing
Hashfunktion soll injektiv sein.
Idee: Zwei Hashfunktionen. Erste verteilt auf Buckets W_i. Jedes Bucket hat eigene Hashfunktion h_i, die in dem Bucket injektiv ist.
Speicher: O(n^2), Lookup: O(1)):
$h_i(x) = (k * x \ mod \ p) \ mod \ s$ mit p: Primzahl > N, s: Bucketanzahl&Größe.
Speicher: O(n), Lookup: O(1):
1. Stufe: Wähle k sodass $\sum_{i=0}^{n-1} |W_i^k|^2 < 3n$. Es gibt n Buckets. Hashfunktion $h_k(x) = (kx \ mod \ p) \ mod \ n$
2. Stufe: Wähle k_i für $h_{k_i}(x)$ so, dass $h_{k_i}(x) = (k_i x \ mod \ p) \ mods_i$ injektiv auf W_i^k ist.

Priority Queues (Fibonacci Heap)

Speichert Paare (p, i) von Prioritäten und Informationen.
Operationen: PQ.insert(p,i), PQ.findmin(), PQ.delmin(),PQ.decreade_p(x,q) (Vermindert $x = (p, i)$ zu (q, i)

Heap ordered Tree: Knoten mit Prioritäten sodass gilt $prio(v) \geq prio(vater(v))$

```
class nod{
        P prio;
        I info;
        node parent,left_sib, right_sib;
        bool mark;
        int rank;
}
```

Damit sind alle Knoten einer Ebene in einer zyklischen Liste. Auch die Wurzeln liegen in einer zyklischen Liste.
Konstruktor für FibHeap:min = 0 (Pointer auf minimale Wurzel)
PQ.findmin(): Zugriff auf min (Minpointer)
PQ.insert(p,i): Erzeuge neuen Baum der nur aus 1 Knoten besteht. Füge ihn in Wurzelliste ein.
PQ.delmin(): v = min. Lösche v aus Wurzelliste, füge alle Kinder ein.Eliminiere Mehrfachvorkommen von Rängen:

```
while ∃ v1,v2 sodass rang(v1) = rang(v2) do
        if not v1.prio ≤ v2.prio then
                vertausche v1,v2
        fi
        Verschmelze(v1,v2) (Haenge Baum groesserer Prio an den kleinerer
od
```

PQ.decrease: Vermindere p, verletzt Heap Eigenschaft. Lösche Pointer zum Vaternknoten und fäge v in Wurzelliste ein. Setze das Löschen in Richtung Wurzel fort, falls ein zweites Kind gelöscht wurde. Aktualisieren minpointer.

Amortisierte Analysen

Datenstruktur D, Potential $pot : D \to \mathbb{R}_0^+$, Operation $op : D' \to^{op} D''$
$T_{\text{tats}}(op)$ = Ausführzeit einer Operation
$T_{\text{amort}}(op) = T_{\text{tats}}(op) + pot(D'') - pot(D') = T_{\text{tats}}(op) + \Delta pot$
$\sum_{i=1}^{n} T_{\text{tats}}(op_i) = \sum_{i=1}^{n} T_{\text{amort}}(op_i) + pot(D_0) - pot(D_n)$ für Folge von Operationen $D_0 \to .. \to D_n$
Problem: Wähle gute Potentialfunktion
Spezialfall: $pot(D_0) = 0, pot(D_i) \geq 0 \Rightarrow \sum_{i=1}^{n} T_{\text{Amort}}(op_i)$
Binärzähler: Sei pot die Anzahl der Einsen bis zur ersten Null im Bitstring (xxxxx0111 = 3 = k)
$T_{\text{tats}}(incr) = O(1+k)$: Alle 1 auf 0 und eine 0 auf 1
$\Delta pot = 1 - k$
$T_{\text{amort}}(incr) = 1 + k + (1 - k) = 2 = O(1)$
Für n Operationen hat die Potentialmethode auf dem Binärzähler Kosten von 2n (O(n))

Djikstra Algorithmus

```
forall v in V do
        DIST[v] = ∞
        PRED[v] = null
od
DIST[s] = 0
PQ.insert(v,0)
while not PQ.empty() do
        u = PQ.delmin()
        forall v in V mit (u,v) in E do
                d = DIST[u] + c(u,v)
                if d < DIST[v] then
```

```
                        if DIST[v] == ∞ then
                                PQ.insert(v,d)
                        else
                                PQ.decrease(v,d)
                        fi
                        DIST[v] = d
                        Pred[v] = u
                fi
        od
od
```

Laufzeit auf Fib-Heap: Insert+Empty+Decrease = O(1), delmin O(log(n))
Bei nicht-negativen Kreisen terminiert Djikstra nicht

Bellman/Ford Algorithmus

U: Schlange von Knoten, in_U[bool] ob Knoten in U, count[int], s Startknoten

```
forall v in V do
        DIST[v] = ∞
        PRED[v] = null
        count[v] = 0
        in_U[v] = false
od
DIST[s] = 0
U.append(s)
in_U[s] = true
while not U.empty() do
        u = U.pop()
        in_U[u] = false
        if ++count[u] > n then
                Print("Negativer Zyklus")
                return
        fi
        forall v in V mit (u,v) in E do
                d = DIST[v] + c(u,v)
                if d < DIST[v] then
                        DIST[v] = d
                        PRED[v] = u
                        if not in_U[v] then
                                U.append(v)
                                in_U[v] = true
                        fi
                fi
        od
od
```

Planare Graphen

Ein ungerichteter Graph ist planar, wenn er eine planare Zeichnung besitzt.
Eine planare Zeichnung eines Graphen ordnet jeden Knoten v einen Punkt $p \in \mathbb{R}^2$ und jeder Kante $\{v, w\} \in E$ eine stetige Kurve mit den Endpunkten Position(v) und Position(w) zu, sodass sich die Kurven paarweise nicht schneiden.
K_5 ist der vollständige Graph mit 5 Knoten, nicht planar
$K_{3,3}$ ist der vollständige bipartite Graph mit 2x3 Knoten, nicht planar
G ist planar $\Leftrightarrow \exists$ planare Zeichnung auf einer Kugeloberfläche (Fläche von Geschlecht 0)
Satz von Kuratowski: G ist planar $\Leftrightarrow$ G enthält keinen K_5 oder $K_{3,3}$

G heißt einfach zusammenhängend, wenn es für jedes beliebige Knotenpaar einen Pfad zwischen diesen gibt.
G heißt zweifach zusammenhängend, wenn G trotz entfernen eines beliebigen Knotens weiterhin zusammenhängend ist. Und so weiter
cut-Vertex/Artikulationspunkt: Wenn dieser Knoten entfernt wird zerfällt der Graph in mehrere Zusammenhangskomponenten.
Maximal planarer Graph: Das hinzufügen einer beliebigen Kante verletzt die Planarität.
Satz von Euler: Für eine planare Einbettung gilt $n - m * f = 2$ mit n Knoten, m Kanten und f Faces
Maximal planarer Graph hat nur Dreiecke $\Rightarrow$ Jede Fläche hat 3 Kanten, eine Kante gehört zu 2 Flächen: $3f = 2m \Rightarrow f = \frac{2}{3}m \Rightarrow n - m + \frac{2}{3}m = 2 \Rightarrow m = 3n - 6$
Bipartite planare Graühen haben nur Vierecke $4f = 2m \Rightarrow f = \frac{1}{2}m = 2 \Rightarrow m = 2n - 4$