

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 -_i
 $hoehe(x) = \log(n)$

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

```
Init:
for i=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 $O(\log(n))$

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 $O(\text{Höhe}(T))$

Insert(x):Bestimme zufällige Priorität. Füge Knoten gemäß Schlüssel ein. Rotiere den Knoten nach oben, bis $prio(\text{vater}(v)) \geq prio(v)$ bzw $v=\text{Wurzel}$

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
        rotate_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 $n/m \geq 4$, verdopple Tafelgröße

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: $O(1)$:

$h_i(x) = (k * x \bmod p) \bmod s$ 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 $h_k(x) = (kx \bmod p) \bmod n$
2. Stufe: Wähle k_i für $h_{k_i}(x)$ so, dass $h_{k_i}(x) = (k_i x \bmod p) \bmod s_i$ 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.decrease_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  $\exists v_1, v_2$  sodass  $rang(v_1) = rang(v_2)$  do
    if not  $v_1.prio \leq v_2.prio$  then
        vertausche  $v_1, v_2$ 
    fi
    Verschmelze( $v_1, v_2$ ) (Haenge Baum groesserer Prio an den kleinerer)
```

od

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

Amortisierte Analysen

Datenstruktur D , Potential $pot : D \rightarrow \mathbb{R}_0^+$, Operation $op : D' \rightarrow^{op} D''$

$T_{\text{tats}}(op)$ = 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 $D_0 \rightarrow \dots \rightarrow D_n$

Problem: Wähle gute Potentialfunktion

Dijkstra 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 Dijkstra nicht

Bellman/Ford Algorithmus

U : Schlange von Knoten, $\text{in_U}[\text{bool}]$ ob Knoten in U , $\text{count}[\text{int}]$, 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[u] + 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