Sei $G=(V,E)$ ungerichteter Graph. Verwenden Sie UNION-FIND zum Aufbau einer DS beschreiben sie jeweils eine Lösung für das Union-Find-Problem mit Laufzeiten
a) $O(\log n)$ (amort.) für UNION & $O(1)$ für FIND
b) $O(1)$ für UNION und $O(\log n)$ für FIND
wobei n #Elt. Begründen Sie in beiden Fällen die entsprechenden Laufzeiten

a) ~~[illegible]~~ Relabel the smaller half

- UNION(A,B) $O(\log n)$

```
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
```

Da immer die kleinere Menge relabelt wird, kann maximal eine Laufzeit von $O(n \log n)$ dafür gebraucht werden. Amortisiert auf n Elt ergibt sich für jedes Union $O(\log n)$

immer wenn x seinen Namen ändert, befindet es sich danach in einer min. doppelt so großen Menge
↳ jedes Elt kann max $\log n$-mal Namen ändern
→ Gesamt $n \log n$ → amort. $\log n$ / höhe(x)

- FIND(x): $O(1)$

```
return name[x]
```

b) Weighted Union Rule: Hierbei gilt ~~dass Lemma~~, dass stets $size(x) \geq 2^{höhe(x)}$ für alle Knoten gilt → höhe(x) = log(n)

- INIT:

```
for i = 1 to n do
    vater[i] = 0     // Vater von i in seinem Baum, o.falls Wurzel
    name[i] = i      // name d. Blocks mit Wurzel i
    wurzel[i] = i    // wurzel des Blocks mit Name i
    size[i] = 1      // # Knoten im Unterbaum mit Wurzel i
od
```

- FIND(x): $O(\log(n))$ ⊙

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

mit Pfadkompression $O(\log(n))$ ⊙

```
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]
```

Sonst nicht.

- UNION(A,B): $O(1)$

```
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
```

Sei $G=(V,E)$ ungerichteter Graph. Verwenden Sie UNION-FIND zum Aufbau einer DS zur Beantwortung von Anfragen SAME-COMP(v,w), die true liefern, wenn v & w in derselben ZHK von G liegen. Wie lange dauert d. Aufbau d. Struktur und was kostet 1 Frage

SAME-COMP(v,w) = { true, v & w in derselben ZHK; false, sonst }

$\alpha(\log n)$ $\mathcal{O}(1)$

INIT: UNION-FIND-Struktur → einelementige Becher. Benutze UNION-FIND-Ops in Variante $\{i\}^m$ $i = 1..n$ mit Relable the smaller half.

```
for all (v,w) ∈ E do
    A ← FIND(v)
    B ← FIND(w)
    if A ≠ B then
        UNION(A,B)
od
```

Kosten für Aufbau

$|E| \cdot$ Kosten für Union $= m \cdot \log n$

$\Rightarrow \mathcal{O}(n \log n)$

```
SAME_COMP(v,w):
return FIND(v) == FIND(w)
```

$\Rightarrow \mathcal{O}(1)$

Geben Sie den Alg. von Dijkstra im Pseudo-Code an & analysieren Sie die Laufzeit

DIJKSTRA (Variante des Grundalgos)

```
forall v ∈ V do
    DIST[v] ← ∞
    PRED[v] ← null
od
DIST[s] ← 0
PQ.insert(u, C)
while not PQ.empty do
    u ← PQ.delmin()   //liefert Info
    forall u ∈ V mit (u,v) ∈ 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: $O(\sum_{v \in V} (1 + outdeg(v)) + PQ\_Operationen)$

$= n \cdot (T_{insert} + T_{delmin} + T_{empty}) +$

$m \cdot T_{decrease}$)

PQ_ops:

$T_{insert} + T_{delmin}$: Jeder Knoten max 1x

$T_{insert} + T_{decrease}$: Innere Schleife

Laufzeit auf Fib-Heap: insert + empty + Decrease = $O(1)$, delmin = $O(\log n)$

Bei neg. Kreisen terminiert Dijkstra nicht ↳ $O(n \cdot \log n + m)$

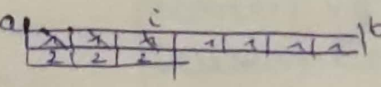
Das Split-Find-Problem ist wie folgt def: Verwalte eine Einteilung der Zahlen $\{1,\dots,n\}$ in disjunkte Intervalle, die am Anfang nur aus Intervall $[1,n]$ besteht unter diesen Ops:
FIND(i): liefert Intervall das Zahl i enthält
SPLIT(i): ersetze Intervall $[a,b]$ = FIND(i) durch die beiden Intervalle $[a,i]$ & $[i+1,b]$
Entwickeln Sie eine DS, die jede FIND-Op in Zeit $O(1)$ und jede Folge von SPLIT-Op möl. eff. unterstützt.

Idee Umkehrung von UNION-FIND.
Feld name$[1,\dots,n]$

INIT: $\forall\, i, 1 \le i \le n$ do
name$[i]=1$
od

FIND(i): return name$[i]$ $\rightarrow O(1)$

SPLIT(i)



- neuer Name des neuen Intervalls das durch split entsteht ++count
- relabel the smaller half

- laufe parallel d.h. abwechselnd nach links und rechts von i aus, bis Intervallgrenze erreicht d.h. name$[i] \neq$ name[betrachtetes Element]
- nenne den Teil um, der kleiner ist $[a,i]$ oder Intervall $[i+1,b]$ indem nochmals über diesen Teil gelaufen wird. name[betrachtetes Element] = count

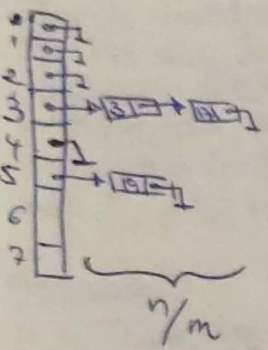
$\Rightarrow$ Kosten für 1 Split $O(2 \cdot$ Länge des kürzeren Intervalls$)$

Analyse $O(\#$ Namensänderungen$)$: mal $\frac{\text{Länge des umzubenennenden Intervalls}}{2}$
$\Rightarrow O(\log n)$

Entwickeln Sie eine DS zur Speicherung von n Schlüsseln aus dem Universum $\{1, \dots, N\}$ (wobei $n \ll N$), die eine Zugriffszeit von $O(1)$ garantiert. Sie dürfen dabei $O(n^2)$ Speicherplatz verwenden

Hashing mit Verkettung: löse Kollisionen nicht auf, speichere mehrere Schlüssel an der gleichen Position

speichere für jedes Ergebnis der Hashfunktion h eine Liste



Lookup(x): lin. Suche in Liste T[h(x)]
wc: alle keys in derselben Liste → O(n)
erw. Zeit: $O(\frac{n}{m})$
Belegungsfaktor $\beta = n/m$ ← erw. Länge einer Liste T[x]
wenn $m \geq n$ d.h $\beta \leq 1$ dann erw. Laufzeit $O(1)$
Insert(x): $x \notin S$. füge x an erste freie Stelle in T[h(x)] ein
Delete(x): Entferne x aus T[h(x)]

m = Bucketanzahl

meist wird als Hashfunktion einfaches Modulo verwendet

Verbesserung Verdopplungs-Strategie
immer wenn $\beta > 2$, verdopple Tafelgröße → 1 sehr teures Insert (da alle Elt mit neuer Hashfunktion umgespeichert werden), im Schnitt weiter $O(1)$

Bei Delete und kleinem β ($\leq \frac{1}{2}$): Tabelle kann halbiert werden → 1 sehr teures Delete, im Schnitt weiter $O(1)$

Perfektes Hashing: Verbessern sie die DS so, dass nur noch Speicherplatz $C(n)$ benutzt wird.

Um perfektes Hashing zu erhalten, verwendet man 2 Stufen von universellem Hashing. Kollisionsvermeidung durch injektive Abbildung auf der 2ten Stufe: für Menge $S \subseteq \{0..N-1\}$, $n=|S|$ verwende Tafel der Größe $m \geq n$ & $m = O(n)$

Idee:
1) verwende randomisiertes Verfahren: wähle Hashfunktion zufällig aus Menge von Kandidaten (Anteil der aus Menge ist mit hoher WK injektiv)
2) 2-Stufiges Hashing-Schema
   s = Tafelgröße auf 1 Stufe
   w = Bucket (enthält Teilmenge von S)
   $w_i = \{x \in S \mid h(x) = i\}$

Umsetzung
1 Stufe: Tafelgröße $s=n$ & wähle k so: $\sum_{i=0}^{n-1} |w_i^k|^2 < 3n$
Genauer: Hashfkt $h_k : x \to (kx \bmod p) \bmod s$ (p eine Primzahl.) verteilt S auf Tafel d. Größe s so dass Summe der Quadrate d. Bucketgrößen $< 3n$ (linear)
2. Stufe: $\forall$ nicht-leere Buckets 1. Stufe: wähle Tafel d. Größe $s_i = 2|w_i^k|^2$ und wähle k_i sodass h_{k_i} injektiv auf S ist. (gilt für min. Hälfte aller k_i)

Analyse
Wir wissen $\exists k$ für 2te Stufe → lange Aufbauzeit $C(n \cdot n)$
wenn man Platz um konstanten Faktor erhöht (bei uns 2) sind 50% aller k_i geeignet → effiziente Aufbauzeit $C(n)$

1. Stufe $3n+1$
2. Stufe: $\sum_{i=0}^{n-1} 2 \cdot |w_i^k|^2 = 2 \cdot \sum_{i=0}^{n-1} |w_i^k|^2 < 10n$.

⇒ insges: $13n + 1 = O(n)$

+ Implementierung?

Beschreiben Sie die Technik der amortisierten Analyse einer Folge von Operationen auf einer DS D. Demonstrieren Sie diese Technik am Bsp. einer Folge von INCREMENT-Operationen auf einem binären Zähler

Amort Analyse: Abschätzung der Kosten einer bel. Folge von Ops auf einer DS D.

$D_0 \xrightarrow{op_1} D_1 \xrightarrow{op_2} \blacksquare \dots \xrightarrow{op_n} D_n$

amort. Kosten durchschnittliche Kosten pro Op.

Bsp.: Binzähler mit den Operationen
- INIT Setze Zähler auf 0 : erzeuge Lok der Länge 1 mit $a_0 = 0$. Kosten $O(1)$
- INCREMENT: erhöht um 1: $a_0 \leftarrow a_0 + 1$; $i \leftarrow 0$; while $a_i = 2$ do $a_i = 0$; $a_{i+1} \leftarrow a_{i+1} + 1$; $i \leftarrow i+1$

Bitstring $\dots a_3\ a_2\ a_1\ a_0$ $\quad a_i \in \{0,1\}$ ← LSB

Zahlwert $\sum_{i \geq 0} a_i 2^i$

Kosten von INCREMENT: $O(1 + \underbrace{\text{Anz. d. Iterationen d. Schleife}}_{\#\text{Übertrag}})$

Potentialmethode:

Potential: $Fkt\ pot : \mathcal{D} \to \mathbb{R}_0^+$

Operation: $op \quad \mathcal{D} \xrightarrow{op} \mathcal{D}''$

Def i) $T_{tats}(op)$ = Ausführungszeit d. Op (d.h. tats. Kosten)

ii) $T_{amort}(op) := T_{tats}(op) + pot(\mathcal{D}'') - pot(\mathcal{D}')$

d.h. $T_{amort}(op) = T_{tats}(op) + \Delta pot$ ← Potentialänderung

Betrachte nun eine Folge von n Operationen

$D_0 \xrightarrow{op_1} D_1 \xrightarrow{op_2} \dots \xrightarrow{op_n} D_n$ (bel. Pot.-Fkt)

Dann gilt: $\sum_{i=1}^{n} T_{amort}(op_i) \overset{Def.}{=} \sum_{i=1}^{n} \left(T_{tats}(op_i) + \underbrace{pot(D_i) - pot(D_{i-1})}_{\Delta pot} \right)$

nur 1 Formel!

$= \sum_{i=1}^{n} T_{tats}(op_i) + pot(D_n) - pot(D_0)$

oder $\sum_{i=1}^{n} T_{tats}(op_i) = \sum_{i=1}^{n} T_{amort}(op_i) + pot(D_0) - pot(D_n)$

Spezialfall $pot(D_0) = 0$, $pot(D_i) \geq 0$ gilt sowieso $\Rightarrow \sum_{i=1}^{n} T_{tats}(op_i) \leq \underbrace{\sum_{i=1}^{n} T_{amort}(op_i)}_{\text{Lässt sich oft gut abschätzen (bei geeigneter Wahl von pot)}}$

↑ Das möchten wir abschätzen

Anwendung auf Zähler: $INIT \to D_0 \xrightarrow{inc} D_1 \xrightarrow{inc} \dots \xrightarrow{inc} D_n$

pot = # Einsen im Bitstring, erfüllt Bed. d. Spezialfalles $pot(D_0) = pot(k) = 0$, $pot(D_n) \geq 0$

1) $T_{tats}(Increment) = 1 + k$ wenn Zähler die Form $\dots 0\underbrace{1\dots1}_{k \text{ Einsen}}$ $k \geq 0$

2) Potentialdifferenz: $\Delta pot = 1 - k$ $\quad 0\underbrace{1\dots1}_{k} \xrightarrow{inc} 1\underbrace{0\dots0}_{k}$

1) und 2) $\Rightarrow T_{amort}(Increment) = \underbrace{1+k}_{tats} + \underbrace{(1-k)}_{\Delta pot} = 2 = O(1)$

$\Rightarrow$ Potentialmethode: Gesamtkosten: $2 \cdot n = O(n)$

Geben Sie den Alg. von Bellman/Ford in Pseudo-Code an & analysieren Sie die Laufzeit

U: Schlange von Knoten, in_U[bool]: ob Knoten in U, count[int]: # 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("Neg. 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
```

Jeder Knoten kann max. n× in Hauptschleife ausgewählt werden

$C(n \sum_{v \in V} (1 + outdeg(v))) = C(n^2 + n \cdot m) = \underline{C(n \cdot m)}$ wenn G zush.hang, dann $m \geq n-1$. In Praxis bis zu $C(n^3)$

Sei G ein planarer Graph mit n Knoten und m Kanten. Folgen Sie aus dem Satz von Euler, dass $m \leq 3n-6$ und dass G einen ~~[illegible]~~ Knoten vom Grad ≤ 5 besitzt

Satz von Euler: Sei G eine planare Einbettung eines zus hängenden Graphen mit n Knoten, m Kanten & f Faces. Dann gilt $n-m+f=2$.

Sei G planarer Graph mit $n \geq 3$ Knoten Dann gilt: $m \leq 3n-6$

Bew: Betrachte mit planaren Graphen mit n Knoten und zeige $m=3n-6$

~~hat planarer Graph ist [illegible] planare E~~

~~Annahme~~

In jeder planaren Einbettung eines mit planaren Graphen sind alle Faces Dreiecke.

$\Rightarrow$ Jede Fläche besitzt genau 3 Kanten & jede Kante gehört zu 2 Flächen

d.h $3 \cdot f = 2 \cdot m \Rightarrow f = \frac{2}{3}m$

Einsetzen in Euler-Formel: $n-m+f=2$

$n - m + \frac{2}{3}m = 2$ /da Graph max. planar

$m = 3n-6$

Jeder planare Graph besitzt einen Knoten v mit $\deg(v) \leq 5$

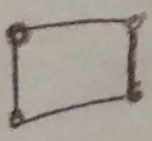
Annahme $\forall v \in V: \deg(v) \geq 6$

$m = \sum_{v \in V} \deg(v)/2$

$\geq 6n/2 = 3n$ $\lightning$ Widerspruch zur Annahme, daraus folgt, dass ein Knoten v ex. muss mit $\deg(v) \leq 5$

Eindeutiger Planarer Graph

Planare Einbettung ist dann eindeutig, wenn diese 3-fach zus. hängend ist



gleicher Graph, versch Einbettungen