Altklausuren Antworten

Landmesser Zusammenfassung

Aufgabe 1

O(log n) (amortisiert) für UNION und O(1) für FIND

Feld name[1...n]:	name[x] = Name des Blocks der x enthält. $1 \leq x \leq n$
size[1..n]:	size[A] = Anzahl Elemente im Block A, initialisiert mit 1
L[1..n]:	L[A] = Liste aller Elemente in Block A, initialisiert L[i] = {i}

```
Initialisierung:
begin
    for i := 1 to n do
        name[i]=i
        size[i]=1
        L[i]={i}
    end
end

FIND(x):
begin
    return name[x]
end

UNION(A,B):
begin
    if size[A] ≤ size[B] then
        foreach i in L[A] do
            name[i] = B
        end
        size[B] += size[A]
        L[B] = L[B].concat(L[A])
    else
        foreach i in L[B] do
            name[i] = A
        end
        size[A] += size[B]
        L[A] = L[A].concat(L[B])
    end
end
```

Laufzeit:

FIND(x): O(1) $\rightarrow$ Einfacher Zugriff auf ein Feld
UNION: O(log n) $\rightarrow$ x kann maximal log(n) mal seinen Namen ändern, da es sich nach jeder Namensänderung in einer doppelt so großen Menge befindet.

O(1) für UNION und O(log n) für FIND

Feld name[1...n]:	name[x] = Name des Blocks mit Wurzel x (hat nur Bedeutung, falls x Wurzel)
Feld vater[1...n]:	vater[x] = $\begin{cases} Vater\ von\ x\ in\ seinem\ Baum \\ 0,\ falls\ x\ Wurzel \end{cases}$
Feld wurzel[1...n]:	wurzel[x] = Wurzel des Blocks mit Namen x

```
Initialisierung:
begin
    for i := 1 to n do
        vater[i]=0
        name[i]=i
        wurzel[i]=i
    end
end

FIND(x):
begin
    while vater[x] != 0 do
        x = vater[x]
    end
    return name[x]
end

UNION(A,B,C):
begin
    r1 = wurzel[A]
    r2 = wurzel[B]
    if size[r1] ≤ 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]
    end
end
```

Laufzeit:

FIND(x): O(log n) $\rightarrow$ Tiefe von x (max Höhe des entstehende Baums, n-1 möglich)

UNION: O(1) $\rightarrow$ Nur Pointer ändern

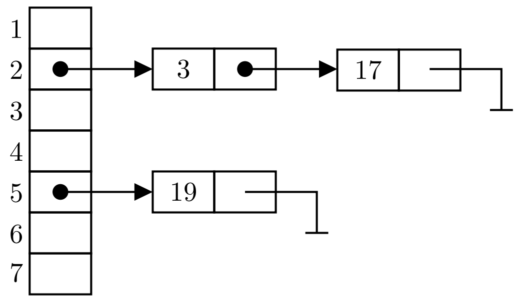
Aufgabe 2

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): lineare Suche in Liste $T[h(x)]$

- Worst Case: alle Keys in derselben List $\rightarrow$ O(n)
- erwartete Zeit: O($\frac{n}{m}$)
- Belegungsfactor $\beta = \frac{n}{m} \leftarrow$ erw. Länge einer Liste T[x]
- wenn $m \geq n$, d.h. $\beta \leq 1$ dann $\rightarrow$ erw. Laufzeit O(1)

Insert(x): $x \notin S$. Füge x an erst freie Stelle in $T[h(x)]$ ein
Delete(x): Entferne x aus $T[h(x)]$



meist wird als Hashfunktion einfaches Modulo verwendet.

Verbesserung Verdopplungs-Strategie:

- Immer wenn $\beta > 2$, verdopple Tafelgröße $\rightarrow$ 1 sehr teures Insert (da alle Elemente mit neuer Hashfunktion umgespeichert werden), im Schnitt aber weiter O(1)

- Bei Delete und kleinem β: Tabelle kann kalbiert werden $\rightarrow$ Ein sehr teures Delte, im Schnitt aber weiter O(1)

Zusatzaufgabe: Perfektes Hashing

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

Idee:

- Verwende randomisiertes Verfahren: Wähle Hashfuntion zufällig aus Menge von Kandidaten (beliebige Funktion aus menge ist mit hoher Wahrscheinlichkeit injektiv)
- 2 Stufiges Hashing-Schema
  - s = Tafelgröße auf 1. Stufe
  - w = Bucket (enhtält Teilmenge von S)
  - $w_i = \{x \in S | h(x) = i\}$

Umsetzung

1. Stufe: Tafelgröße $s = n$ und wähle k so, dass $\sum\limits_{i=0}^{n-1} |w_i^k|^2 < 3n$
Genauer:
Sei $p \in Prim$. Die Hashfunktion $h_k(x) : x \leftarrow (k * x \mod p) \mod s$ verteilt S auf Tafel der Größe s so, dass Summe der Quadtrate der Bucketgrößen $< 3n$ (linear)
2. Stufe: $\forall$ nicht leere Buckets 1. Stufe: Wähle Tafel der Größe $s_i = 2|w_i^k|^2$ und wähle k_i so, dass h_{k_i} injektiv auf S ist. (Gilt für mindestens die Hälfte aller k_i

Analyse:

Wir wissen $\exists k$ für 2. Stufe $\rightarrow$ lange Aufbauzeit ($C(N * n)$) wenn man Platz um konstanten Faktor erhöht (bei uns 2) sind 50% der k's geeignet $\Rightarrow$ effiziente Aufbauzeit ($C(n)$)

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

$\Rightarrow 13n + 1$

Aufgabe3

Amortisierte Analyse:
Abschätzung der Kosten einer beliebigen Folge von Operationen auf der Datenstruktur D.
$D_0 \rightarrow^{op_1} D_1 \rightarrow^{op_2} ... \rightarrow^{op_n} D_n$

Amortisierte Kosten:
durchnittliche Kosten pro Operation

Definition:

- $T_{tats}(op_i)$:= Ausführungszeit von op_i, tatsächliche Kosten
- Potenzial $pot : D \rightarrow \mathbb{R}$ ordnet jedem Zustand von D eine reelle Zahl zu.

- $T_{amort}(op_i)$ ammortisierte Kosten von $op_i := T_tats(op_i) + \underbrace{pot(D_i) - pot(D_{i-1})}_{\Delta_{pot} Potenzialdifferenz}$

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

$$\Rightarrow \underbrace{\sum_{t=1}^{n} T_{tats}(op_i)}_{\text{Wollen wir abschätzen}} = \underbrace{\sum_{t=1}^{n} T_{amort}(op_i)}_{\text{Können wir oft leicht abschätzen}} + pot(D_0) - pot(D_n)$$

Meist verwendet man ein Potential mit:

- $pot(D_0) = 0$
- $pot(D_i) \geq 0$

$\Rightarrow \sum_{t=1}^{n} T_{tats}(op_i) \leq \sum_{t=1}^{n} T_{amort}(op_i)$

Bei Wahl einer geeigneten Potentialfunktion gilt für $\sum_{t=1}^{n} T_{amort}(op_i)$ somit:

- oft leicht abschätzbar
- liefert gute Schranke

Beispiel: Binärzähler

Sei:

- $x \in \mathbb{N}_0$ in Binärdarstellung $\rightarrow x = ...a_2, a_1, a_0$ mit $a_i \in \{0, 1\}$
- $pot(x)$ Anzahl der Einsen in Binärdarsellung von x.
  Erfüllt: $pot(D_0) = 0 \wedge pot(D_i) \geq 0$

Init:

Setze alle a_i's auf 0

Increment um 1:

$a_0 = a_0 + 1;$
$i = 1;$
while $a_i = 2$ **do**
  $a_i = 0;$
  $a_{i+1} = a_{i+1} + 1;$
  $i = i + 1;$
end

Analyse:

- $T_{tats}(Inc) = 1 + k$, wenn x die Form ...0 $\underbrace{1...1}_{k \text{ Einsen}}$ hat, mit $k \geq 0$
- Potenzialdifferentz: $\Delta pot = 1 - k$ $\quad (0\underbrace{1...1}_{k} \rightarrow^{Inc} 1\underbrace{0...0}_{k})$

$\Rightarrow T_{amort}(Inc) = \underbrace{(1+k)}_{tats} + \underbrace{(1-k)}_{\Delta pot} = 2 = O(1)$
$\Rightarrow$ Potenzialmethode: Gesamtkosten: $2n = O(n)$

Aufgabe 4

Sei G ein planarer Graph mit $n \geq 3$ Knoten und m Kanten, dann gilt $m = 3n - 6$. dh $m = O(n)$, also linear viele Kanten.

H. Ein maximal planarer Graph ist ein planarer Graph, der durch Hinzufügen einer Kante $(v, w) \notin E$ nicht-planar wird. Beobachtung: Alle Faces in jeder planaren Einbettung von G sind Dreiecke (Triangulierung). Jedes Face in einer Triangulierung hat 3 Rand-Kanten und jede Kante liegt am Rand von 2 Faces.

$$\Rightarrow 3f = 2m$$

$$\Rightarrow f = \frac{2}{3}m$$

Einstetzen in Euler-Formel

$$n - m + \frac{2}{3}m = 2$$

$$m = 3n - 6$$

$\Rightarrow m \leq 3n - 6$ für beliebige planare Graphen $\square$

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

Annahme:

$\forall_{v \in V} : deg(v) \geq 6$

$$\begin{aligned} \Rightarrow m &= \sum_{v \in V} * \frac{deg(v)}{2} \\ &= \frac{6n}{2} \\ &= 3 \end{aligned}$$

$\lightning$ zur Annahme $\Rightarrow \exists_{v \in V} : deg(v) \leq 5$

Zusatzaufgabe

Sei G ein **bipartiter** planarer Graph Dann gilt $m \leq 2n - 4$.

Beweis: Keine Kreise ungerader Länge in bipartiten Graphen. Kleinstmögliche Fläche in einem bipartiten Graphen ist ein Viereck. $\square$

Aufgabe 5

Split-Find-Problem Idee - Unkehrung von Union-Find Feld name[1,...,n]

Initialisierung:
begin
  for $\forall i, 1 \leq i \leq n$ **do**
    | name[i] = 1
  end
end

Find(i) begin
| return name[i] $\rightarrow \mathcal{O}(1)$
end
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 $[i+1, b]$ indem nochmals über diesen Teil gelaufen wird. name[betrachtetes Element] = count

a			i			b
$\cancel{1}$	$\cancel{1}$	$\cancel{1}$	1	1	1	1
2	2	2				

$\Rightarrow$ Kosten für 1 Split $\mathcal{O}(2 * \text{Länge des kürzeren Intervalls})$
Analyse: $\mathcal{O}(\#\text{Namensänderungen}) : max(\frac{\text{Länge des umzubennenendes Intervall}}{2}) \Rightarrow \mathcal{O}(\log n)$

Aufgabe 6

Algorithmus von Dijkstra:

```
begin
    foreach v ∈ V do
        DIST[v]← ∞
        PRED[v]←NULL
    end
    DIST[s]←0
    PQ.insert(v,0)
    while not PQ.empty() do
        u ← PQ.delmin() //liefertInfo
        foreach v ∈ 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)
                end
                else
                    PQ.decrease(v,d)
                end
                DIST[v]←d
                PRED[v]←u
            end
        end
    end
end
```

Laufzeitanalyse:

Normal:

$\mathcal{O}(\Sigma_{v \in V}(1 + outdeg(v)) + PQ_{Ops})$
$PQ_{Ops} : n * (T_{insert} + T_{delmin} + T_{empty}) + m * T_{decrease}$
$T_{insert} + T_{delmin}$: Jeder Knoten max 1x Innere Schleife

Fibonacci-Heap:

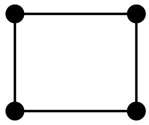
Amortisierte Analyse ist ok, da Gesamtlaufzeit betrachtet.
$\mathcal{O}(n * log(n) + m)$, insert+empty = $\mathcal{O}(1)$, delmin=$\mathcal{O}(log(n))$, decrease = $\mathcal{O}(1)$

Binärer Heap

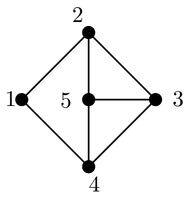
Bei Binärem Heap: $\mathcal{O}(n * log(n) + m * log(n) = \mathcal{O}((n + m) * log(n)$

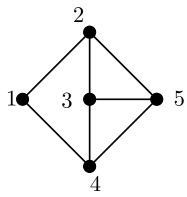
Aufgabe 7

Eine Planare Einbettung ist genau dann eindeutig wenn diese 3-fach zusammenhängend ist:



Gleicher Graph verschiedene Einbettungen:





Aufgabe 8