Zusammenfassung für EAA

Wintersemester 2013/2014

von Dagmar Sorg

Divide and Conquer

1 MergeSort

1.1 Laufzeit

1. Aufteilung der n Elemente in zwei Instanzen mit $\left\lceil \frac{n}{2} \right\rceil$ und $\left\lfloor \frac{n}{2} \right\rfloor$ Elementen
2. rekursive Lösung des Problems
3. Laufzeit von Merge ist **linear**
4. es gibt Konstanten c_1, c_2, sodass die Laufzeit der folgenden entspricht:
   $T(n) \leq T(\left\lceil \frac{n}{2} \right\rceil) + T(\left\lfloor \frac{n}{2} \right\rfloor) + c_2 \cdot n(\text{falls } n > 1), T(1) = c_1$

2 Substitutions-Methode

Raten einer Laufzeit mit Beweis durch Induktion

2.1 Raten durch Ähnlichkeit

sehen, dass eine Rekursionsformel asymptotisch ähnlich ist wie eine andere

2.2 Raten durch Verändern der Variablen

Beispiel $(T(n) = 2T(\sqrt{n}) + \log n)$**:** $n = 2^m, S(m) = T(2^m) = 2 \cdot T(2^{\frac{m}{2}}) + m = 2 \cdot S(\frac{m}{2}) + m$
$\Rightarrow S(\frac{m}{2}) \in O(m \log m)$
$\Rightarrow$ Rücksubstitution: $T(n) \in O(\log n \log \log n)$

2.3 Induktionsbehauptung stärker machen

wenn die Annahme richtig ist, aber die Induktionsvorraussetzung zu schwach ist

Beispiel $\left(T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1\right)$**:** Annahme: $T(n) \in \mathcal{O}(n)$
$\Rightarrow T(n) = c \cdot \left\lfloor \frac{n}{2} \right\rfloor + c \cdot \left\lfloor \frac{n}{2} \right\rfloor = cn + 1,$
aber das heißt noch nicht, dass $T(n) \leq cn$.
Wir nehmen das folgende an:
$T(n) \leq c \cdot \left\lfloor \frac{n}{2} \right\rfloor - b + c \cdot \left\lceil \frac{n}{2} \right\rceil - b + 1 = cn - 2b + 1 \leq cn - b$, falls $b \geq 1$.

3 Iterative Methode

Iteratives Lösen von Rekursionsgleichungen, sodass die Rahmenbedingungen stimmen

Beispiel $\left(T(n) = \begin{cases} c_1 & \textbf{falls } n \leq 3 \\ 3 \cdot T(\left\lfloor \frac{n}{4} \right\rfloor) + c_2 \cdot n & \textbf{sonst} \end{cases}\right)$**:**

$$\begin{aligned} T(n) &= 3 \cdot T(\left\lfloor \frac{n}{4} \right\rfloor) + c_2 \cdot n \\ &= 3 \cdot \left(3 \cdot T(\left\lfloor \frac{n}{16} \right\rfloor) + c_2 \cdot n \left\lfloor \frac{n}{4} \right\rfloor\right) + c_2 \cdot n \\ &= 3 \cdot \left(3\left(3 \cdot T(\left\lfloor \frac{n}{64} \right\rfloor) + c_2 \cdot n \left\lfloor \frac{n}{16} \right\rfloor\right) + c_2 \cdot \left\lfloor \frac{n}{4} \right\rfloor\right) + c_2 \cdot n \\ &= c_2 \cdot \sum_{i=0}^{k-1} 3^i \left\lfloor \frac{n}{4^i} \right\rfloor + 3^k T\left(\left\lfloor \frac{n}{4^k} \right\rfloor\right) \end{aligned}$$

Die Randbedingungen gelten, falls $\frac{n}{4^k} < 4$, bzw. falls $k > \log_4 n - 1$ für das kleinste k. Somit erhalten wir

$$\begin{aligned} T(n) &\leq c_2 \cdot \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + c_1 \cdot 3^{\log_4 n} \\ &\leq 4c_2 \cdot n + c_1 \cdot n^{\log_4 3} \\ &\leq (4c_2 + c_1) \cdot n \end{aligned}$$

$\Rightarrow T(n) \in \mathcal{O}(n)$

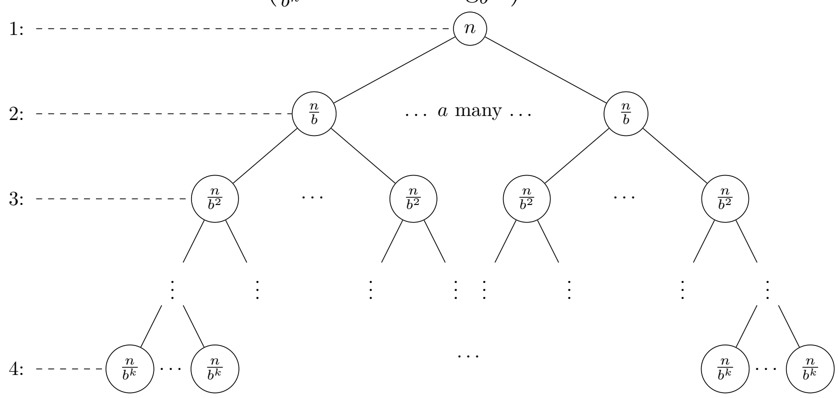
4 Master Methode (Master Theorem)

a) generelle Lösung für Rekursionsformeln der Form
$T(n) = a \cdot T(\frac{n}{b}) + f(n)$

b) $a, b \geq 1$ sind Konstanten

c) $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$

d) erste Annahme: $n = b^k$ $\left(\frac{n}{b^k} = 1 \Leftrightarrow k = \log_b n\right)$:



1. $f(n)$

2. $f(n) + a \cdot f\left(\frac{n}{b}\right)$

3. $f(n) + a \cdot f\left(\frac{n}{b}\right) + a^2 \cdot f\left(\frac{n}{b^2}\right)$

4. $f(n) + a \cdot f\left(\frac{n}{b}\right) + a^2 \cdot f\left(\frac{n}{b^2}\right) + c_0 \cdot a^k$ (wobei $k \approx \log_b n$)

Endsumme: $c_0 \cdot \underbrace{a^{\log_b n}}_{n^{\log_b a}} + \sum_{i=0}^{\log_b n - 1} a^i \cdot f\left(\frac{n}{b^i}\right)$

e) somit gilt in Rekursionsschritt i: zusätzlicher Aufwand von $a^i f\left(\frac{n}{b^i}\right)$

f) falls in Rekursionstiefe k der Wert $\frac{n}{b^k}$ klein genug ist, kann er durch die Konstante c_0 ersetzt werden

4.1 Laufzeit

$$T(n) = c_0 \cdot \underbrace{a^{\log_b n}}_{n^{\log_b a}} + \sum_{i=0}^{\log_b n - 1} a^i \cdot f\left(\frac{n}{b^i}\right)$$

4.2 Laufzeitbestimmung mit dem Master Theorem

$a \geq 1, b > 1, \epsilon > 0, f : \mathbb{N} \to \mathbb{R}_{\geq 0}$, sowie $T(n) = a \cdot T(\frac{n}{b}) + f(n)$ ($\frac{n}{b}$ ist entweder $\left\lfloor \frac{n}{b} \right\rfloor$ oder $\left\lceil \frac{n}{b} \right\rceil$)

Fall 1: Voraussetzung: $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon})$ für beliebiges $\epsilon > 0$

Folgerung: $T(n) \in \mathcal{O}(n^{\log_b a})$

Beispiel: $T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$

$\Rightarrow a = 8, b = 2, f(n) = 1000n^2, \log_b a = \log_2 8 = 3$
$\Rightarrow 1000n^2 \in \mathcal{O}\left(n^{3-\epsilon}\right)$

Fall 2: Voraussetzung: $f(n) \in \Theta\left(n^{\log_b a}\right)$

Folgerung: $T(n) \in \Theta\left(n^{\log_b a} \log n\right)$

Beispiel: $T(n) = 2T\left(\frac{n}{2}\right) + 10n$

$\Rightarrow a = 2, b = 2, f(n) = 10n, \log_b a = \log_2 2 = 1$
$\Rightarrow 10n \in \Theta\left(n^1\right)$

Fall 3: Voraussetzung: $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ für ein $\epsilon > 0$ und falls die Regularitätsbedingung gilt (ein c mit $0 < c < 1$: $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$)

Folgerung: $T(n) \in \Theta(f(n))$

Beispiel: $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

$\Rightarrow a = 2, b = 2, f(n) = n^2, \log_b a = \log_2 2 = 1$
$\Rightarrow n^2 \in \Omega\left(n^{1+\varepsilon}\right)$

Regularitätsbedingung: $2\left(\frac{n}{2}\right)^2 \leq c \cdot n^2 \Leftrightarrow \frac{1}{2}n^2 \leq cn^2$
$\Rightarrow T(n) \in \Theta(n^2)$

5 Anwendung

5.1 Matrix Multiplikation

Problem: Multiplikation zweier $n \times n$ Matrizen

Eingabe: Matrizen $A, B \in \mathbb{R}^{n \times n}$

$$\overbrace{\begin{pmatrix} a_{11} & \dots & a_{1j} & \dots & a_{1n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i1} & \dots & a_{ij} & \dots & a_{in} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nj} & \dots & a_{nn} \end{pmatrix}}^{A} \cdot \overbrace{\begin{pmatrix} b_{11} & \dots & b_{1j} & \dots & b_{1n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{i1} & \dots & b_{ij} & \dots & b_{in} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nj} & \dots & b_{nn} \end{pmatrix}}^{B} = \overbrace{\begin{pmatrix} c_{11} & \dots & c_{1j} & \dots & c_{1n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ c_{i1} & \dots & c_{ij} & \dots & c_{in} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nj} & \dots & c_{nn} \end{pmatrix}}^{C}$$

Ausgabe: Matrix C

Laufzeit: $n^3 + n^2(n-1) \in \Theta(n^3)$

Idee zur Verbesserung der Laufzeit 1 (Divide and Conquer):

1. Aufteilung der Matrizen in 4 $\frac{n}{2} \times \frac{n}{2}$ Matrizen $\Rightarrow C_{ij} = A_{i1} \cdot B_{1j} + A_{i2} \cdot B_{2j}, 1 \leq i, j \leq 2$
2. Laufzeit:
   $T(n) = 8T\left(\frac{n}{2}\right) + 4 \cdot n^2$
   $\overset{\text{Master Theorem (1)}}{\Rightarrow} \Theta(n^3)$

Idee zur Verbesserung der Laufzeit 2 (Strassen):

1. Multiplikation von nur sieben Matrizenpaaren, sowie nur 18 Additionen von Matrizen (Idee: Merken von berechneten Werten)
2. Laufzeit:
$$T(n) = \begin{cases} n^3 + n^2(n-1) & \text{falls } n \leq 2^{k_0} \text{ für eine Konstante } k_0 \geq 0 \\ 7T\left(\frac{n}{2}\right) + 18 \cdot \left(\frac{n}{2}\right) & \text{sonst} \end{cases}$$
$\overset{\text{Master Theorem (1)}}{\Rightarrow} \Theta(n^{\log_2 7}) \subset \mathcal{O}(n^{2.91})$ (wobei n eine Zweierpotenz ist)

Beste asymptotische Laufzeit: Bei einem ALgorithmus von Coppersmith und Winograd (1990): $\mathcal{O}(n^{2.37\cdots})$. Es gibt auch Algorithmen mit einer geringeren asymptotischen Laufzeit, aber mit riesigen Konstanten.

Amortisierte Analyse

Ein Algorithmus kann aus mehreren Operationsabfolgen bestehen. Hier kann man eine obere Grenze der Worst-Case-Laufzeit bestimmen, indem man die Worst-Case-Laufzeit einer Operation nimmt und sie mit der Anzahl an Operationen multipliziert. Die wirkliche Worst-Case-Laufzeit kann jedoch besser sein.

Beispiel ($MultiPop$)**:**

Push($element$**):** $element$ wird dem Stack hinzugefügt

MultiPop(k**):** k Elemente werden vom Stack geholt (wenn weniger als k Elemente auf dem Stack sind, werden alle geholt)

1 Accounting Methode (Abrechnungsverfahren)

1. Idee: Bezahlen für mögliche kommende Operationen mithilfe von amortisierten Kosten $\hat{c}$
2. $c - \hat{c}$ (c sind die wirklichen Kosten) sind die reservierten Kosten für spätere Operationen, dessen $\hat{c}$ nicht für die wirklichen Kosten ausreichen
3. für $\hat{c}$ gilt: $\sum\limits_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$ und ist somit eine obere Grenze der Gesamtkosten

Beispiel ($MultiPop\ (Fortsetzung)$)**:**

1. aktuelle Kosten für Push: 1 Einheit
2. aktuelle Kosten für MultiPop: $\min(k, |S| + 1)$
3. amortisierte Kosten für Push: 2 Einheit (1 für Push, die andere für MultiPop)
4. amortisierte Kosten für MultiPop: 1 Einheit (benötigt, falls $k > |S|$)

Alle Kosten sind konstant $\Rightarrow$ Laufzeit ist linear (in $\mathcal{O}(n)$)

2 Potentialfunktionsverfahren

1. definieren einer Potentialfunktion Φ, die jedem möglichen Zustand einer Datenstruktur einen Wert zuweist
2. bei einer Abfolge von n Operationen erhalten wir: $\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\text{Potentialdifferenz}}$

   mit D_i ist Zustand der Datenstruktur nach der i-ten Operation und D_0 Startzustand vor der ersten Operation
   $\Rightarrow \sum\limits_{i=1}^{n} c_i = \sum\limits_{i=1}^{n} \hat{c}_i + \Phi(D_0) - \Phi(D_n)$
3. wenn Φ so gewählt ist, dass $\Phi(D_n) \geq \Phi(D_0)$, dann ist $\sum\limits_{i=1}^{n} \hat{c}_i$ eine Obergrenze der Gesamtkosten des Algorithmus.

Beispiel ($MultiPop\ (Fortseztzung)$)**:**

1. Φ ist die Anzahl $|S|$ der Elemente auf dem Stack S
2. amortisierte Kosten von Push: $\hat{c} = 1 + \Phi(D_1) = 1 + 1 = 2$
3. amortisierte Kosten von MultiPop(k): $\hat{c} = \min(k, |S| + 1) - \min(k, |S|) \in \{0, 1\}$

Somit ist die Laufzeit linear ($\in \mathcal{O}(n)$).

UNION-FIND-DATENSTRUKTUR

1. es wird eine endliche Menge X verwendet
2. Ziel: dynamische Menge $\mathcal{S}$ von disjunkten Teilmengen von X
3. vorhandene Methoden:

   MakeSet(item x): erstellt eine neue Menge nur mit dem Item x ($\{x\}$)

   Find(item x): gibt die Menge mit dem Item x zurück

   Union(set i, set j): erstellt eine neue Menge mit den Mengen i, j und löscht die beiden Mengen i, j
4. an kann annehmen dass $X = \{1, \ldots, n\}$ mit $n \in \mathbb{N}$ ist, da man für andere Mengen jedem Item eine einzigartige Zahl zuordnen kann
5. jede Menge hat einen **Repräsentanten**, FINDgibt diesen zurück, UNIONbekommt diese als Argumente

Im Folgenden betrachten wir eine Sequenz mit m Operationen MAKESET, FIND und UNION, wobei n die Anzahl an MAKESET-Operationen ist.

1 Array Darstellung

Beispiel ($\mathcal{S} = \{\{1,3,5,7\},\{2,4,8\}\}, X = \{1,\ldots,9\}$):

Item x	1	2	3	4	5	6	7	8	9
Menge $A[x]$	1	2	1	2	1	0	1	2	0

Laufzeiten:

MakeSet: $\Theta(1)$

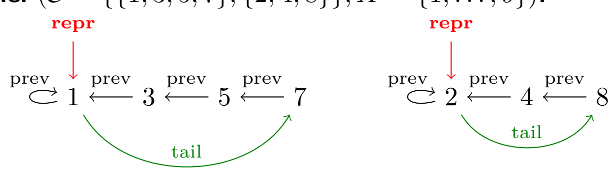
Find: $\Theta(1)$

Union: $\Theta(n)$

2 LinkedList Darstellung

Zur Reduzierung der Laufzeit von UNION

Beispiel ($\mathcal{S} = \{\{1,3,5,7\},\{2,4,8\}\}, X = \{1,\ldots,9\}$):



Laufzeiten:

MakeSet: $\Theta(1)$

Find: $\Theta(n)$

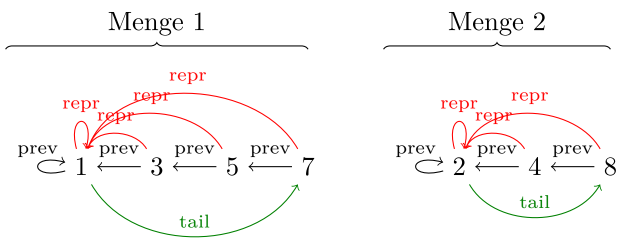
Union: $\Theta(1)$

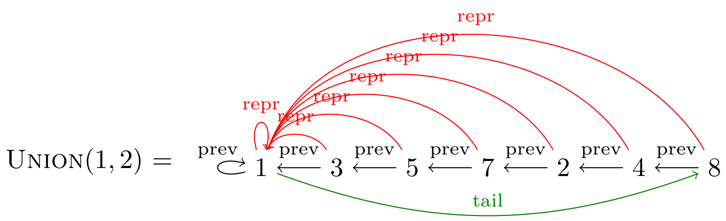
Gesamtlaufzeit für $n-1$ Union und m Find: $\Theta(m \cdot n)$

$\Rightarrow$ keine Verbesserung der Laufzeit

2.1 Erweiterte LinkedList Darstellung

Beispiel ($\mathcal{S} = \{\{1,3,5,7\},\{2,4,8\}\}, X = \{1,\ldots,9\}$):





Wenn man die Länge jeder Liste speichert und immer die kürzere Liste an die Längere hängt, wird jeder Repräsentanten-Zeiger höchstens $\lfloor \log n \rfloor$-mal verändert werden.
Laufzeit von einer Sequenz mit m Operationen (MakeSet, Union, Find) liegt in $\mathcal{O}(m + n \log n)$

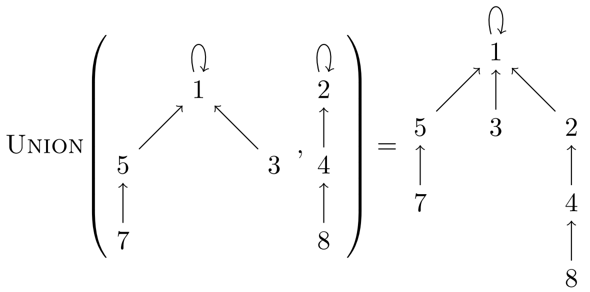
3 Rooted Tree Darstellung

Repräsentant: Wurzel des zugehörigen Baumes

Union(a, b)**:** Anhängen der Wurzel von a an Wurzel von b

Find(a)**:** Aufsteigen im Baum bis zur Wurzel von a

Beispiel (**Union**$(1, 2)$):



Laufzeiten:

MakeSet: $\Theta(1)$

Union: $\Theta(1)$

Find: Die Laufzeit von Find ist anhängig von der Höhe des Baumes. Wenn Union einfach ohne Überprüfung der Höhe der Bäume durchgeführt wird, liegt Find in $\Theta(n)$.

3.1 gewichtete Vereinigung (weighted Union)

Es wird der kleinere Baum an den größeren angehängt. Damit das möglich ist, wird die Größe jedes Baumes folgendermaßen gespeichert: $parent[root] = -size$.
Wenn ein Baum aus mehreren *weighted* Union Operationen entstanden ist, so gilt: $h(T) \leq \log |T|$, wobei $h(T)$ die Höhe des Baumes und $|T|$ die Anzahl der Elemente in T ist.

Baum T_j wurde an Baum T_i angehängt. Dann gilt: $h(T) = \max(h(T_j) + 1, h(T_i))$. Somit entstehen zwei Fälle:

1. $h(T_i) > h(T_j) + 1 \Rightarrow h(T) = h(T_i) \leq \log |T_i| < T$ ✔
2. $h(T_i) \leq h(T_j) + 1$
   $\Rightarrow h(T) = h(T_j) + 1 \leq \log |T_j| + 1 = \log(2 \cdot |T_j|) \leq \log(|T_j| + |T_i|) = \log |T|$ ✔

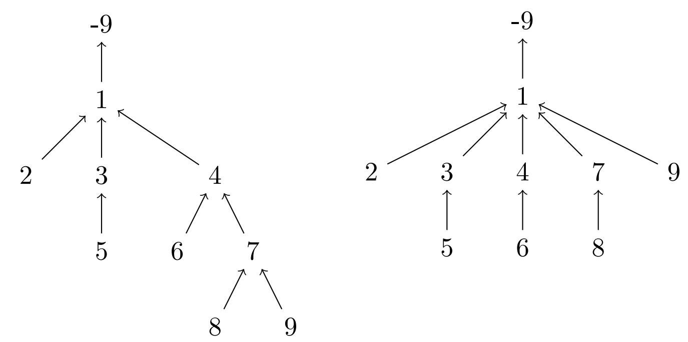
$\Rightarrow$ Eine Sequenz von n MakeSet-Operationen und m *weighted* Union- und Find-Operationen, kann in $\mathcal{O}(m \log n)$ ausgeführt werden.

3.2 Find mit "Path Compression"

Bei der Suche nach dem Schlüssel k ändern wir für alle Knoten auf dem Pfad von *root* zu a den Zeiger zum Vorgänger ($parent[x] \leftarrow root$, x liegt auf dem Pfad von *root* zu a).

Beispiel (**Find**(9))**:**

Vor der Suche nach 9: Nach der Suche nach 9:



Laufzeiten:

Find: $\Theta(\log n)$

Union: $\Theta(1)$

MakeSet: $\Theta(1)$

Mit der Anwendung der amortisierten Kosten erhält man jedoch folgendes:

Find: $\Theta(\log^* n)$

Wobei folgendes gilt (**iterativer Logarithmus**):

$$\log^* n = \min\{j \geq 0; \log^{(j)} n \leq 1\}$$

sowie

$$\log^{(i)} n = \begin{cases} n & \text{falls } i = 0 \\ \log(\log^{(i-1)} n) & \text{falls } i > 0 \text{ und } \log^{(i-1)} > 0 \text{ definiert} \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Der **rank** $r(v)$ eines Knotes v entspricht der Höhe seines Teilbaumes, gewurzelt bei v. Somit gilt

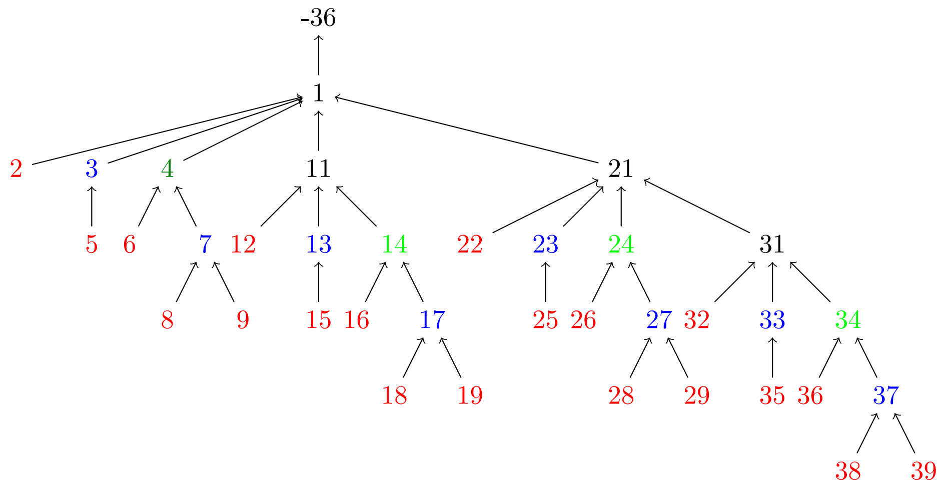
$$r(v) \leq \log n, \ \forall v \in V$$

Eine **Rank**-Gruppe R_j ist eine Menge von Knoten für die gilt:

$$R_j = \begin{cases} \{v | \log^{(j+1)} n > r(v) \leq \log^{(j)} n\} & \text{falls } \log^{(j+1)} n \text{ definiert ist} \\ \{v | r(v) = 0\} & \text{falls } \log^{(j)} n < 1 \text{ definiert ist} \\ \emptyset & \text{sonst} \end{cases}$$

Beispiel ($r(1) = 5, r(21) = 4, r(11) = r(31) = 3$, **grüne:** $r() = 2$, **blaue:** $r() = 1$, **rote:** $r() = 0$)**:**

Sowie R_1 sind die schwarzen Knoten,
R_2 sind die grünen Knoten,
R_3 sind die blauen Knoten,
R_4 sind die roten Knoten.



Alle ranks steigen zur jeder Zeit der Sequenz auf dem Weg eines Knotens zur Wurzel strikt monoton an (auf einem Pfad vom Knoten zur Wurzel).

Beweis:

Zu einem bestimmten Punkt setzen wir für einen Knoten v: $parent[v] \leftarrow w$ durch die Pfadkompression (davor war v in einem Teilbaum von w). Somit war vorher schon $r(v) < r(w)$.

Es gibt höchstens $\frac{n}{2^r}$ Knoten vom rank r.

Beweis:

T_v ist Teilbaum gewurzelt bei v vom rank r im Wald T'. Dann gilt

$$r = h(T_v) \leq \log |T_v| \quad \Rightarrow |T_v| \geq 2^r$$

Da zwei Teilbäume mit der selben rank disjunkt sind und es insgesamt n Knoten gibt folgt daraus, dass es höchstens $\frac{n}{2^r}$ Knoten pro rank gibt.

Beginn der amortisierten Analyse:

1. Originalsequenz (σ)
2. Hinzurechnen der Kosten einer Operation $\textsc{Find}(x)$ zu der Operation für das Bewegen der Knoten (eine Einheit für das Durchlaufen der Knoten auf einem Pfad x zur Wurzel (inklusive x, ohne Wurzel und Vorgänger der Wurzel) und eine Einheit für das Bewegen der Knoten)
3. zwei Arten von Bewegungen:

   Typ A: Vor der Bewegung gilt $R_i(v), R_j(parent[v]), i \neq j$

   Typ B: Vor der Bewegung gilt $R_i(v), R_j(parent[v]), i = j$
4. es gibt höchstens $\log^* n + 1$ nicht-leere Rank-Gruppen
5. weil der rank eines Knotens auf dem Weg zur Wurzel ansteigt folgt, dass es höchstens $\log^* n$ Bewegungen vom *Typ A* gibt
6. es gibt weniger als $\log^j n$ Bewegungen in der Rank-Gruppe R_j
7. es gibt höchstens $\frac{n}{2^r}$ Knoten pro rank

Hieraus folgt:

$$\begin{aligned} |R_j| &< \sum_{i=\lceil \log^{(j+1)} n \rceil}^{\infty} \frac{n}{2^i} \\ &= \frac{n}{2^{\lceil \log^{(j+1)} n \rceil}} \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &\leq \frac{2n}{2^{\log^{(j+1)} n}} \\ &= \frac{2n}{2^{\log(\log^{(j)} n)}} \\ &= \frac{2n}{\log^{(j)} n} \end{aligned}$$

Somit gibt es $|R_j| \cdot \log^{(j)} = 2n$ Bewegungen vom Typ B pro Rank-Gruppe.
$\Rightarrow 2n \cdot \log^* n + 1$ Bewegungen vom Typ B.
Zusammenfassend:
Eine Sequenz von m Operationen MakeSet, gewichtete Union und Find mit Pfadkompression (n sind MakeSet-Operationen) kann in $\mathcal{O}(m \log^* n)$ ausgeführt werden.

3.3 inverse Ackermannfunktion

Wächst langsamer als der iterative Logarithmus, die m Operationen können in $\mathcal{O}(m\alpha(m,n))$ ausgeführt werden, wobei α eine Variante der inversen Ackermannfunktion ist.

4 Anwendung: Gleichheit von endlichen Automaten

- **witness** ist ein Beispiel, das zeigt, dass zwei Automaten nicht gleich sind.
- zwei Automaten können nur dann gleich sein, wenn ihre Startzustände gleich sind
- zwei Automaten sind gleich, wenn sie die gleiche Menge an Wörtern akzeptieren
- Algorithmus zum Testen der Gleichheit von endlichen Automaten kann dann eine **kürzeste** *witness* ausgeben, wenn die Datenstruktur zum Speichern der Zustände als Queue und nicht als Stack realisiert wird (ansonsten kann auch eine längere *witness* ausgegeben werden)
- der Algorithmus ist korrekt, weil alle möglichen Wege gespeichert und somit überprüft werden
- Laufzeit: es kann in $\mathcal{O}(|\Sigma| \cdot (|Q_1| + |Q_2|) \cdot \log^*(|Q_1| + |Q_2|)$ entschieden werden, ob zwei Automaten gleich sind oder nicht

Minimaler Spannbaum

inzident:

- ein Knoten v und eine Kante e sind inzident, falls $v \in e$
- zwei Kanten e_1, e_2 sind inzident, falls $e_1 \cap e_2 \neq \emptyset$

adjazent: zwei Knoten v, w sind adjazent, falls $\{v, w\} \in E$

Grad: $deg(v) = \#$ inzidenter Kanten

Pfad der Länge l: ist ein Teilgraph mit allen Kanten des Pfades mit $l + 1$ Knoten

verbundener Teilgraph: ist ein maximal verbundener Teilgraph (alle Kanten zwischen den Knoten $v \in V_{Teilgraph}$ sind in $E_{Teilgraph}$)

Baum: $m = n - 1$ und ist verbunden

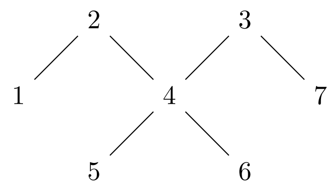
gespannter Teilgraph: ist ein verbundener Teilgraph mit $V_{Teilgraph} = V$

gespannter Teilbaum: ist ein gespannter Teilgraph, der ein Baum ist

1 Prüfer-Sequenz

Es gibt n^{n-2} beschriftete Bäume auf der Knotenmenge $\{1, \dots, n\}$ für alle $n \in \mathbb{N}_{\geq 1}$.
Ein Baum T kann definiert werden durch $T = \text{Prüfer2Tree}(\text{Tree2Prüfer}(T))$
Beispiel (Prüfer-Sequenz: $(2, 4, 4, 4, 3)$):

Baum:



Zurück zum Baum:

j	1	2	5	6	4	7
$parent[j]$	2	4	4	4	3	3

2 Tarjan's Kantenfärbungs-Methode

- **Farbeninvariante:** Es gibt einen MST, der alle blauen und keine rote Kante enthält.
- eine Kante $e = \{v, w\} \in E$ **kreuzt** einen *Schnitt*, falls $v \in S \subsetneq V$ und $w \in V \setminus S$
- ein **einfacher Kreis** ist ein verbundener (Teil-)Graph mit $\forall v \in V : deg(v) = 2$
- wenn T ein Spannbaum ist, so gibt es für jeden Schnitt in G eine Kante, die diesen Schnitt kreuzt, sowie es in jedem Kreis eine Kante gibt, die nicht in T ist

Blaue Regel: Auswählen eines Schnittes, den keine blaue Kante kreuzt $\rightarrow$ färbe Kante mit dem kleinsten Gewicht blau

Rote Regel: Auswählen eines einfachen Kreises, der keine rote Kante enthält $\rightarrow$ färbe die Kante mit dem größten Gewicht rot

Dieser Algorithmus wird solange angewendet, bis keine Regel mehr angewendet werden kann.

Tarjan's Kantenfärbungsalgorithmus färbt alle Kanten richtig.
Beweis:

Am Anfang ist keine Kante gefärbt. Da der Graph verbunden ist, gibt es auch einen MST. Nach dem k-ten Schritt gibt es einen MST T mit allen blauen und keinen roten Kanten. Jetzt gibt es zwei Fälle:

Anwendung der blauen Regel: Falls der Algorithmus eine Kante $e \in T$ färbt, ist alles ok. Sonst gibt es eine Kante e' auf dem Schnitt $C = (S, V \setminus S)$ die nicht blau gefärbt ist und zu T gehört (sie kann nicht rot sein, sonst wäre sie nicht im Baum T). Dann färben wir die Kante e blau. Da immer die Kante mit dem kleinsten Gewicht genommen wird, gilt $w(T') \leq w(T)$.

Anwendung der roten Regel: Äquivalent zur blauen Regel mit einem Kreis C sowie der Folgerung, dass $w(e) \geq w(e')$ und $w(T') \leq w(T)$.

Zum zeigen, dass der Algorithmus auch alle Kanten färbt müssen wir folgende zwei Fälle zeigen:

$e \in T$: Betrachten der beiden Komponenten, die durch den Schnitt C durch e entstehen: keine blaue Kante geht über C, somit können wir e blau färben.

$e \notin T$: Betrachten den Kreis C (der einzigartige Pfad von v nach w, wobei $e = \{v, w\}$), dann gibt es keine rote Kante auf C und wir können die rote Regel anwenden.

3 Kruskal's Algorithmus

- wird mit n blauen disjunkten Bäumen gestartet
- Kanten werden in nicht-absteigender Reihenfolge (bezogen auf ihr Gewicht) abgearbeitet
- falls eine Kante e inzident zu zwei Knoten in **verschiedenen** Bäumen ist, wird die Kante **blau** gefärbt, sonst rot
- Anwendung der Färbungsregeln von Tarjan
  Beweis:
  Falls e in zwei unterschiedlichen blauen Bäumen endet, kann man S als die Menge an Knoten definieren, die v enthält. Dann kreuzt keine blaue Kante den Schnitt $C = (S, V \setminus S)$ und durch das Ordnen der Kanten ist e die Kante mit dem geringsten Gewicht.
  Falls $e = \{v, w\}$ inzident zu zwei Knoten im selben Baum ist, ist der Pfad P zwischen v und w zusammen mit e ein einfacher Kreis ohne rote Kanten. Somit wird e rot gefärbt (e ist die einzige ungefärbte Kante).
- Laufzeit:
  - Sortieren der Kanten in $\mathcal{O}(m \log n)$
  - Union-Find-Datenstruktur in $\mathcal{O}(m \log^* n)$
  - Gesamtlaufzeit somit in $\mathcal{O}(m \log n)$

4 Matroids und der Greedy Algorithmus

5 Der Algorithmus von Prim

Fibonacci-Heaps

Minimaler Cut in ungerichteten Graphen

Network Flows und minimale Cuts

Tipps