Algorithmen und Datenstrukturen 1

ALGO1 · SoSe-2023 · tcs.uni-frankfurt.de/algo1/ · 2023-07-06 · *ed6968a*

Disjunkte Mengen, Union-Find (Woche 8)

Eigenständige Vorbereitung:
Lies 📖 CLRS Kapitel 21 ohne 21.4 (oder Algorithms 4ed. Kapitel 1.5) und schau dir das 📺 Video der Woche an.

Zeichenlegende:

- ✏️ Schriftliche Aufgabe, die du fristgerecht in Moodle abgibst. In der Klausur wirst du alle Aufgaben schriftlich bearbeiten, daher ist das Feedback der Tutoren wichtig, damit du deine Schreibfähigkeiten verbessern kannst.
- 👍 Diese Art von Aufgabe musst du sicher können, um die Klausur zu bestehen.
- 🔑 Diese Art von Aufgabe musst du weitgehend können, um die Klausur zu bestehen.
- 🔭 Diese Art von Aufgabe musst du können, um eine gute Note zu erhalten.
- 🌈 Diese Aufgabe ist als Knobelspaß gedacht, der das algorithmische Verständnis vertieft.

Aufgabe 8.1 (Union Find von Hand laufen lassen 👍). Betrachte die folgende Sequenz von Operationen: `Init(7)`, `Union(3,4)`, `Union(5,0)`, `Union(4,5)`, `Union(4,3)`, `Union(0,1)`, `Union(2,6)`, `Union(0,4)` und `Union(6,0)`.

a) (einfach) Führe die Sequenz mittels ***Quick Find*** von Hand durch. Zeige, wie die Inhalte des Feldes `id` nach jedem Schritt aussehen. Die Operation `Union`(i, j) verändert `id` hierbei immer für die Menge, die durch i gegeben ist.

b) (einfach) Führe die Sequenz mittels ***Quick Union*** von Hand durch. Zeige, wie der Baum nach jedem Schritt aussieht. Die Operation `Union`(i, j) setzt hierbei immer die Wurzel von Baum von i als ein Kind der Wurzel des Baumes von j.

c) Führe die Sequenz mittels ***Weighted Quick Union*** von Hand durch. Zeige, wie der Baum nach jedem Schritt aussieht. Die Operation `Union`(i, j) setzt hierbei immer die Wurzel von Baum von i als ein Kind der Wurzel des Baumes von j, wenn die Größe der beiden Bäume gleich ist.

d) Zeige das Resultat der ***Pfadverkürzung*** nach einer Operation `Find`(x) in einem der Bäume aus den Beispielen in a) und b), wobei x einmal ein Blatt sein soll, einmal ein interner Knoten von Tiefe 1, und einmal ein interner Knoten mit Höhe 1.

e) Gib eine Sequenz von Operationen an, die in einem Baum maximaler Tiefe resultieren, wenn die abstrakte Datenstruktur durch ***Quick Union*** implementiert ist.

f) Gib eine Sequenz von Operationen an, die in einem Baum maximaler Tiefe resultieren, wenn die abstrakte Datenstruktur durch ***Weighted Quick Union*** implementiert ist.

g) Schreibe den Pseudo-Code für ***Pfadverkürzung***. *Hinweis: Durchlaufe den Pfad zweimal.*

Aufgabe 8.2 (Alternative zum *Quick Find* Algorithmus 🔑). Eine Kommilitonin stellt die folgende, intuitive Variante von ***Quick Find*** `Union` vor. Funktioniert sie?

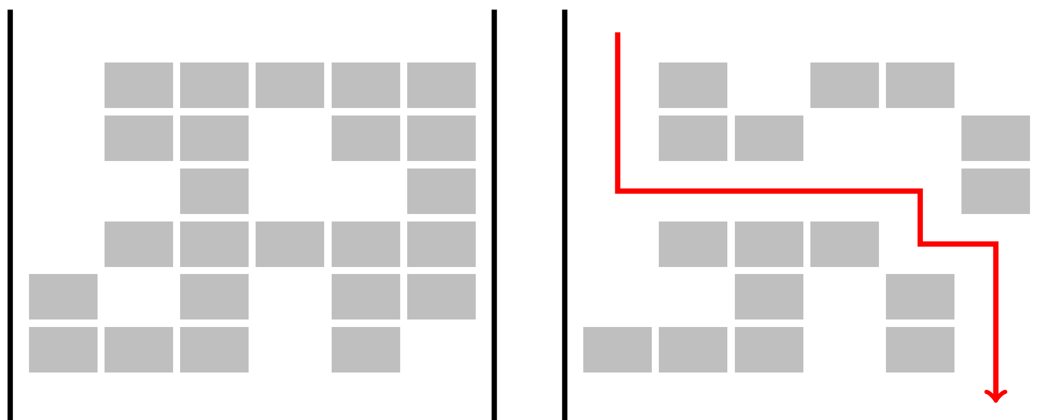
```
procedure UNION(i, j)
    if FIND(i) ≠ FIND(j) then
        for all k ∈ {0, ..., n − 1} do
            if id[k] == id[i] then
                id[k] = id[j]
```

Aufgabe 8.3 (Dynamische Zusammenhangskomponente und Suche in Graphen 🔑).
Mit Tiefen- und Breitensuche können wir die Zusammenhangskomponenten eines Graphen finden. Entwirf eine einfache Implementierung der abstrakten Datenstruktur für den dynamischen Zusammenhang (mit den Operationen INIT, CONNECTED, INSERT) mittels Suche in Graphen und vergleiche die Komplexität deiner Lösung mit der auf Union-Find basierenden Lösungen.

Aufgabe 8.4 (Implementierung von Union-Find, alleine probieren 🔑). Wir wollen Datenstrukturen für Union-Find (mit den Operationen `Init`, `Union` und `Find`) in einer beliebigen Programmiersprache implementieren.

a) Implementiere ***Quick Union***.
b) Erweitere deine Implementierung mit ***Weighted Quick Union***.
c) Erweitere deine Implementierung mit ***Pfadverkürzung***.

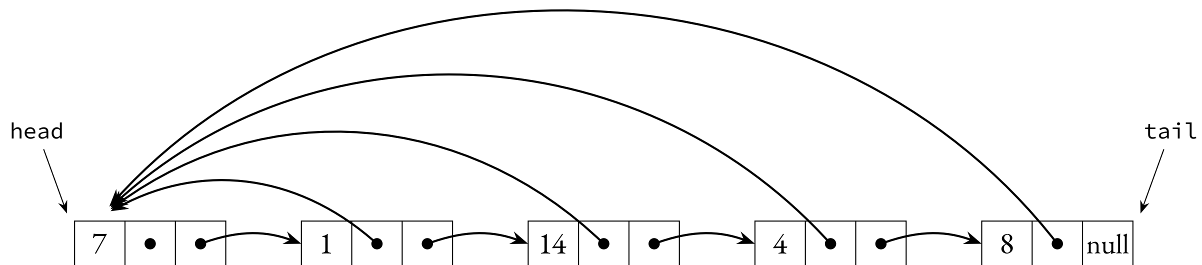
Aufgabe 8.5 (Zombieinvasion 🌈). Der erkenntnisgierige Prof. Dr. Regloh hat bei unethischen Zombieduellexperimenten versehentlich einige Zombies entkommen lassen und damit die postapokalyptische Zukunft eingeläutet. Du hast dich mit einer kleinen Gruppe an Überlebenden in einem kleinen Gebäude verbarrikadiert. Das Einzige, das zwischen euch und einer brutalen und hungrigen Horde Zombies steht, ist eine starke Befestigung. Die Befestigung besteht aus einem $k \times k$ Gitter von Wänden, hier illustriert durch ein 6×6 Gitter von Wänden (*graue Rechtecke*):



Oberhalb des Gitters warten die Zombies, während deine Gruppe und du unterhalb sind. Unglücklicherweise sind die Wände alt und spröde, und stürzen regelmäßig ein. Sobald ein Pfad von ganz oben nach ganz unten verläuft, kommen die Zombies durch. Um eure Evakuierung vorzubereiten, willst du durchgehend überwachen, ob es derzeit einen Pfad durch die Befestigung gibt. Entwirf eine Datenstruktur, die effizient den Überblick behält, während die Wände (Zellen des Gitters) eine nach der anderen einstürzen.

Aufgabe 8.6 (Rekursive *Pfadverkürzung* 🔭). Entwirf eine rekursive Variante von ***Pfadverkürzung*** in Pseudo-Code.

Aufgabe 8.7 (Union-Find mit verketteten Listen und Gewichtungen). Wir wollen eine Variante von ***Quick Find*** mittels verketteten Listen auf die folgende Art und Weise implementieren. Jede Menge wird durch eine einfach verkettete Liste repräsentiert. Der Repräsentant jeder Menge ist das erste Element der jeweiligen Liste und jedes Element der Liste hat einen Zeiger auf den Repräsentanten. Des Weiteren haben wir einen Zeiger `tail` auf das letzte Element der Liste. Zum Beispiel könnte die Datenstruktur für die Menge $\{1, 4, 7, 8, 14\}$ mit Repräsentant 7 wie folgt aussehen:



a) 🔑 Zeige wie man mit dieser Darstellung der Mengen die Operationen `Init`(n) in Zeit $O(n)$, `Find`(i) in Zeit $O(1)$ und `Union`(i, j) in Zeit $O(|S(i)|)$ implementieren kann, wobei $S(i)$ diejenige Menge ist, die i enthält.

b) 🔑 Zeige, wie man die Lösung erweitern kann, sodass `Init` und `Find` dieselbe Zeitkomplexität wie zuvor haben, aber `Union`(i, j) nun nur Zeit $O(\min(|S(i)|, |S(j)|))$ benötigt. *Hinweis: Speichere ein paar Zusatzinformationen.*

c) 🔭 Zeige, dass für die Lösung aus b) jede Sequenz von p `Find` und m `Union` Operationen auf n Elementen eine Laufzeit von $O(p + m \log n)$ hat. Zu Beginn der Sequenz sind alle Mengen einelementig.

Aufgabe 8.8 (Union Find Move ✏️). Entwickle eine **möglichst effiziente** Implementierung der folgenden abstrakten Datenstruktur: Verwalte eine Familie von disjunkten Mengen, anfangs die einelementigen Mengen

$$\{0\}, \{1\}, \dots, \{n-1\},$$

unter den folgenden Operationen:

0 („query“) Die *query* Operation nimmt zwei Argumente s und t, und ermittelt, ob s und t zur selben Menge gehören. Das heißt, wenn $s \in S$ and $t \in T$, dann gibt die Operation `1` aus falls $S = T$ und `0` wenn $S \neq T$.

1 („union“) Die *union* Operation nimmt zwei Argumente s und t, und erzeugt die Vereinigung der beiden Mengen, die s und t enthalten. Das heißt, wenn $s \in S$ und $t \in T$ mit $S \neq T$, dann sollen S und T aus der Familie entfernt und durch die Menge $S \cup T$ ersetzt werden. (Wenn $S = T$, dann passiert nichts.)

2 („move“) Die *move* Operation nimmt zwei Argumente s und t, und verschiebt das Element s in die Menge, die t enthält. Das heißt, wenn $s \in S$ und $t \in T$ mit $S \neq T$, dann sollen S und T aus der Familie entfernt werden und stattdessen die Mengen $S - \{s\}$ (falls diese nichtleer ist) und $T \cup \{s\}$ zur Familie hinzugefügt werden. (Wenn $S = T$, dann passiert nichts.)

Diese Operationen erhalten die Invariante, dass die Mengen in der Familie disjunkt sind.

a) 👍 Beschreibe eine naive Datenstruktur, bei der jede Operation bis zu $O(n)$ Zeit brauchen darf.
oder: 🔑 Beschreibe eine deutlich effizientere Datenstruktur und analysiere sie.

b) 👍 Implementiere und teste deine Datenstruktur gemäß der Anforderungen in Abschnitt A.

Hinweise zur Abgabe. Die Union-Find Datenstrukturen aus der Vorlesung reichen hier nicht aus, sondern müssen modifiziert werden, daher ist eine klare und anschauliche Beschreibung der Idee besonders wichtig. Kleine Bildchen wären nützlich, um den Zustand deiner Datenstruktur zu verstehen. Erwartet werden ansonsten wie üblich die grobe Idee, Pseudocode oder echter Code, Korrektheitsbeweis, Laufzeitanalyse.

A. Implementierung

Für die Implementierung soll folgendes Ein- und Ausgabeverhalten unterstützt werden.

Eingabe. Die Eingabe startet mit zwei natürlichen Zahlen n und m in der ersten Zeile. Die Zahl n ist die Anzahl an einelementigen Mengen zu Beginn. Dann folgen m Zeilen der Form „`0` s t“ (für *query*) oder „`1` s t“ (für *union*) oder „`2` s t“ (für *move*). Du kannst $0 \leq s < n$ und $0 \leq t < n$ annehmen.

Ausgabe. Für jede *query* Operation wird wie oben beschrieben eine Zeile mit `1` oder `0` ausgegeben.

Beispiel.

example.in	Zustand der Datenstruktur nach der Operation
`4 9`	$\{0\}, \{1\}, \{2\}, \{3\}$
`0 0 1`	
`1 0 1`	$\{0, 1\}, \{2\}, \{3\}$
`0 0 1`	
`1 1 2`	$\{0, 1, 2\}, \{3\}$
`0 1 2`	
`0 0 3`	
`2 2 3`	$\{0, 1\}, \{2, 3\}$
`0 0 1`	
`0 1 2`	

example.ans

```
0
1
1
0
1
0
```

Mehr Tests. Hier gibt es größere Tests: https://files.tcs.uni-frankfurt.de/algo1/unionfindmove-tests.zip.

Stelle sicher, dass deine Implementierung auf allen Instanzen exakt die richtige Ausgabe liefert. Die Datei `032-huge.in` hat noch keine Antwort und soll gelöst werden; die meisten naiven Datenstrukturen können diese Instanz vermutlich nicht lösen. Eine effiziente Lösung dagegen braucht auf einem handelsüblichen Laptop nicht lange:

```
$ time python oursolution.py < 032-huge.in > /dev/null

________________________________________________________
Executed in    8.85 secs    fish           external
   usr time    8.79 secs  822.00 micros    8.79 secs
   sys time    0.06 secs  309.00 micros    0.06 secs
```

Liefere die SHA-1 Summe von `032-huge.ans`.