

Grundlagenpraktikum: Rechnerarchitektur**Zusatzaufgabe 8: Union Find****1 Einführung**

Datensätze können oft in disjunkte Gruppen anhand verschiedener Kriterien unterteilt werden. Für diverse Algorithmen ist es notwendig, effiziente Abfragen zu diesen Gruppen auch für große Datensätze zu ermöglichen. Einer dieser Anwendungsfälle ist die Suche nach minimalen Spannbäumen auf Graphen, die für eine ganze Reihe an Anwendungsfällen interessant sind. In dieser Aufgabe untersuchen wir die Union-Find Datenstruktur, die uns erlaubt diverse Abfragen mit guter asymptotischer Laufzeit zu bearbeiten.

Sei n die Anzahl an Datenpunkten. Ein Datenpunkt wird durch einen Integer $D_i \in [0, n)$ eindeutig identifiziert. Eine Gruppe, bestehend aus beliebig vielen Datenpunkten, wird ebenso durch einen Integer $G_i \in [0, n)$ eindeutig identifiziert. Weiterhin gibt es für jeden Gruppe G_i einen Rang R_i .

In der Union-Find Datenstruktur werden eine Vielzahl an Bäumen (ein sogenannter Wald) abgespeichert, wobei alle Knoten eines Baumes derselben Gruppe angehören.

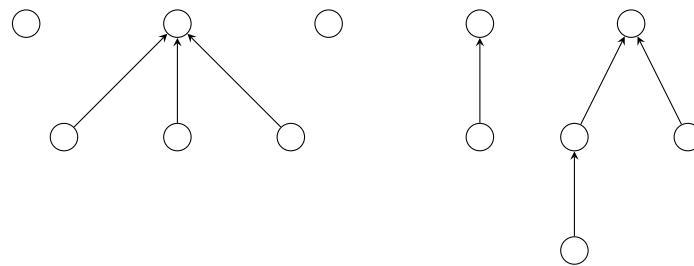


Abbildung 1: Ein Wald aus Bäumen, wie man ihn in einer Union-Find Datenstruktur finden könnte. Ein Baum kann durchaus nur aus einem Knoten bestehen.

Die Union-Find Datenstruktur unterstützt zwei Arten von Operationen: Find und Union. Im Folgenden schauen wir uns diese genauer an.

Find

$\text{Find}(D_i) \rightarrow G_i$: Die Find Operation (Abbildung 2) berechnet die Gruppenzugehörigkeit eines Datenpunktes D_i . Sie gibt G_i zurück, einen eindeutigen Identifier der Gruppe. Wir nennen G_i den Repräsentanten von allen Datenpunkten in dieser Gruppe. Der Repräsentant der Gruppe ist äquivalent zu der Wurzel von dem Baum, der Datenpunkt D_i enthält. Die Find Operation traversiert also den Baum von Knoten D_i aufwärts bis zur Wurzel G_i und gibt G_i zurück.

Relinking: Während der Traversierung des Baumes werden alle Knoten auf dem Suchpfad, die nicht direkt unter dem Repräsentanten (der Wurzel) hängen, eine Ebene höher direkt unter ihrem Großelternknoten neu eingehängt. Dies verringert die Tiefe der Bäume und verbessert damit die asymptotische Laufzeit. Mehr dazu im Beispiel.

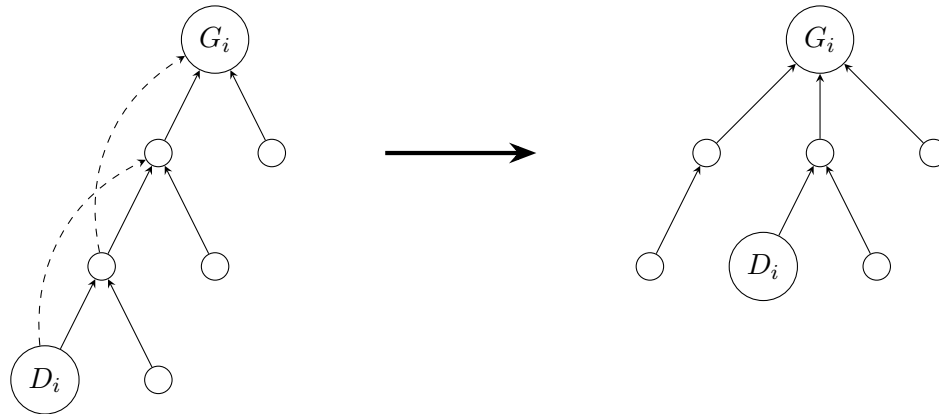


Abbildung 2: Das Relinking während einer Find Operation. Auf dem Suchpfad von D_i nach G_i finden wir mehrere Knoten, die nicht direkt unter G_i hängen. Sobald G_i gefunden wurde, hängen wir diese Knoten direkt unter ihre ursprünglichen Großelternknoten.

Mit der Find Operation lässt sich zum Beispiel überprüfen, ob zwei Datenpunkte D_1 , D_2 derselben Gruppe zugehören (äquivalent: Es gibt im Graphen einen Pfad von D_1 zu D_2). Dies ist der Fall genau dann, wenn $\text{Find}(D_1) == \text{Find}(D_2)$.

Union

$\text{Union}(D_i, D_j) \rightarrow G_{i/j}$: Die Union Operation (Abbildung 3) vereint in der Datenstruktur die beiden Gruppen $G_i = \text{Find}(D_i)$ und $G_j = \text{Find}(D_j)$ miteinander zu einer größeren Gruppe. Dies passiert, indem einer der Repräsentanten (und damit dessen Baum) in den anderen Baum direkt unter dem anderen Repräsentanten eingehängt wird. Zurückgegeben wird der Repräsentant der vereinten Gruppe.

Um zu entscheiden, welcher Repräsentant der neue Repräsentant der Gesamtgruppe wird, wird der Rang R_i verwendet. Dabei wird immer die Gruppe mit dem niedrigeren Rang in die Gruppe mit dem größeren Rang absorbiert. Der Rang ist eine obere Schranke für die Baumtiefe:

1. Wenn beide Gruppen dieselbe Baumtiefe aufweisen, dann ist die Reihenfolge der Argumente ausschlaggebend, das heißt es wird G_j in G_i eingehängt. Dabei erhöht sich die Baumtiefe bei G_i um 1 und dementsprechend wird der Rang bei G_i um 1 inkrementiert.

2. Wenn $R_i > R_j$, dann wird G_j in G_i eingehängt. Wir wissen, dass R_j (die maximal mögliche Tiefe des Baumes bei G_j) zwingend kleiner ist als R_i (die maximal mögliche Tiefe des Baumes bei G_i). Daher wird der Baum unter G_i durch das Einhängen von G_j nicht tiefer, der Rang erhöht sich somit nicht.
3. Für $R_i < R_j$ gilt Fall 2 analog.

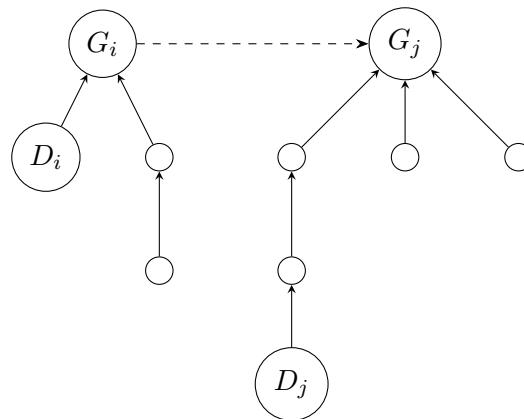







Abbildung 3: $\text{Union}(D_i, D_j)$. Weil der Baum von G_j tiefer ist als der von G_i wird G_i an G_j angehängt.





Edge case: Der Fall, dass G_i und G_j gleich sind (das heißt D_i und D_j befinden sich bereits in einer Gruppe), muss gegebenenfalls gesondert behandelt werden.

2 Umsetzung & Beispiel




Für die Implementierung der Union-Find Datenstruktur verwenden wir eine Array-Darstellung des Waldes. Dafür benötigen wir zwei Arrays. Im ersten Array wird der jeweilige Index des Elternknotens eines Datenpunktes im Baum gespeichert. Wenn der Knoten kein Elternknoten hat (und dementsprechend ein Repräsentant ist), dann speichern wir den eigenen Index im Array ab um die Repräsentanten identifizieren zu können. Ein Repräsentant lässt sich also finden, indem wiederholt der Elternknoten des aktuellen Knotens nachgeschlagen wird. Im zweiten Array wird der zugehörige Rang des Baums unter dem aktuellen Knoten (inklusive dem Knoten selbst) abgespeichert. Zu Beginn ist jeder Datenpunkt der Repräsentant seiner eigenen Gruppe und alle Ränge sind 1.

| | | | | | |
|-------------|---|---|---|--|---|
| Index D_i | 0 | 1 | 2 | 3 | 4 |
| Group G_i | 0 | 1 | 2 | 3 | 4 |
| |  |  |  |  |  |
| Rank R_i | 1 | 1 | 1 | 1 | 1 |



Wenn wir nun ein Union zwischen D_0 und D_1 ausführen, dann wird D_1 in die Gruppe von D_0 absorbiert. Da beide den gleichen Rang haben muss der Rang bei D_0 angepasst werden, da der Baum (eventuell) tiefer geworden ist. Beim anderen Knoten wird der Rang nicht mehr benötigt.

| | | | | | |
|-------------|---|---|---|--|---|
| Index D_i | 0 | 1 | 2 | 3 | 4 |
| Group G_i | 0 | 0 | 2 | 3 | 4 |
| |  | |  |  |  |
| Rank R_i | 2 | | 1 | 1 | 1 |

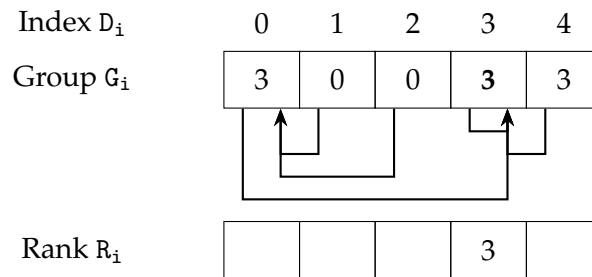
Wir können nun zum Beispiel ein Union zwischen D_1 und D_2 ausführen. Wir Find-en dafür die Repräsentanten von D_1 und D_2 und aktualisieren wieder deren Indices. Der Rang erhöht sich nicht.

| | | | | | |
|-------------|---|---|---|--|---|
| Index D_i | 0 | 1 | 2 | 3 | 4 |
| Group G_i | 0 | 0 | 0 | 3 | 4 |
| |  | | |  |  |
| Rank R_i | 2 | | | 1 | 1 |

Nun können wir mit einem Union D_3 und D_4 diese beiden Gruppen zusammenfügen.

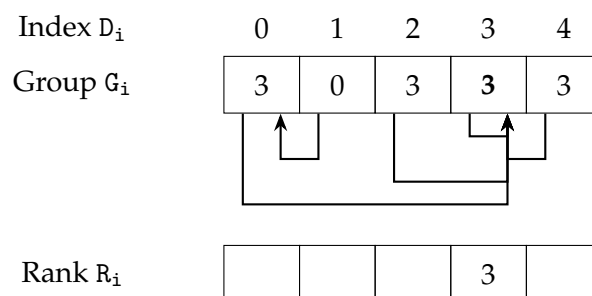
| | | | | | |
|-------------|---|---|---|--|---|
| Index D_i | 0 | 1 | 2 | 3 | 4 |
| Group G_i | 0 | 0 | 0 | 3 | 3 |
| |  | | |  | |
| Rank R_i | 2 | | | 2 | |

Schließlich können wir zum Beispiel Union D_4 und D_2 ausführen. Die Ränge sind identisch, daher spielt die Reihenfolge der Parameter eine Rolle. Da wir immer nur die Repräsentanten der Gruppen verändern, schaut unsere Datenstruktur nun so aus:



Nun zeigt D_2 nicht mehr direkt auf den Repräsentanten, sondern nur noch indirekt. Unser Baum ist dementsprechend tiefer geworden, und erhält somit einen größeren Rang. Wenn wir jetzt $\text{Find}(D_2)$ ausführen, dann muss Find erst von Element 2 zu Element 0, und anschließend von Element 0 zu Element 3 suchen. Bei Element 3 wurde dann der Repräsentant gefunden, da $D_i = G_i$.

Um den Baum möglichst flach zu halten, aktualisieren wir die Elemente auf dem Suchpfad von Find mittels Relinking. Alle Elemente auf dem Suchpfad, die nicht den Repräsentanten als Elternknoten haben, bekommen dabei ihren Großelternknoten als neuen Elternknoten. Im Beispiel ist der einzige Knoten auf dem Suchpfad, der nicht direkt unter dem Repräsentanten hängt D_2 . Dieser bekommt nun den Elternknoten seines eigenen Elternknotens, also D_3 als neuen Elternknoten. Danach sieht die Datenstruktur so aus:



Wenn wir nun erneut $\text{Find}(D_2)$ ausführen, dann finden wir den Repräsentanten schneller. Mit diesem Relinking in der Find Operation können wir auf einen Baum von n Knoten eine Sequenz von m Union oder Find Operationen mit einer Laufzeitkomplexität von $\mathcal{O}(m * \alpha(n))$ ausführen, wobei α die überaus langsam wachsende inverse Ackermannfunktion ist.

Notiz: Beim Relinking wird der Rang nicht verändert obwohl der Baum eventuell weniger tief geworden ist. Der Rang ist somit nur eine Obergrenze für die Baumtiefe.

Level

Neben dem tatsächlichen Resultat der Operationen Find und Union interessiert es uns auch noch, wie schnell unsere Operation durchgeführt wurde. In beiden Operationen gibt es nur eine Teiloperation, die nicht in konstanter Laufzeit durchgeführt werden

kann: Die Suche nach dem Wurzelknoten und das anschließende Relinking. Daher würden wir gerne Wissen, wie viele Knoten unserer Algorithmus in jeder Operation inspizieren musste, bevor die Wurzel gefunden wurde. Diese Kennzahl nennen wir die Anzahl an traversierten Leveln. Ist der erste inspizierte Knoten direkt die Wurzel, dann ist die Anzahl an traversierten Leveln 0.

3 Aufgabe

Eure Aufgabe ist es, die Union-Find Datenstruktur in x86-64 Assembler zu implementieren. Die Funktionssignatur für eure Implementation ist wie folgt:

```
1 void unionfind(
2     uint64_t setSize,
3     char* instruction_string,
4     char* solution_string);
```

Die Variable `setSize` ist n , die Anzahl der Datenpunkte D_i . Der `instructionString` ist ein ASCII String, der eine Sequenz an Union/Find Anweisungen (als 'U' und 'F' kodiert) enthält. In `solutionString` schreibt ihr einen ASCII String mit den Resultaten der Operationen.

Beispielein/ausgaben

| setSize | instructionString | solutionString | Notiz |
|-----------|-----------------------------|-------------------------------------|--------------------|
| $n = 3$ | "F0 F1 F2" | "F0L0 F1L0 F2L0" | Startkonfiguration |
| $n = 2$ | "U0&1 F0 F1" | "U0L0+ F0L0 F0L1" | Union + 2×Find |
| $n = 5$ | "U0&1 U1&2 U3&4 U4&2 F2 F2" | "U0L0+ U0L1 U3L0+ U3L2+ F3L2- F3L1" | Beispiel von oben |
| $n = 999$ | "F42" | "F42L0" | Ein einzelner Find |

Tabelle 1: Diverse Beispiel Ein- und Ausgabestrings der Referenzimplementierung

Die Anweisung 'U' (Union) nimmt 2 Operanden, die mit einem '&' getrennt sind. Die Anweisung 'F' (Find) nimmt nur einen Operanden. Der `instructionString` ist eine Aneinanderreihung von beliebig vielen dieser Anweisungen, getrennt durch Leerzeichen.

Für jede Anweisung hängt ihr folgendes an den `solutionString`: die Art der Anweisung (U/F), das Resultat (der Repräsentant der Gruppe), ein Literal 'L', sowie die Gesamtanzahl an traversierten Leveln, die während der Ausführung der Anweisung angefallen sind¹. Wenn während der Operation mindestens ein Baum komprimiert wurde, ist zusätzlich ein '-' anzuhängen. Ist ein Rang während einer Operation gewachsen, wird ein '+' angehängt. Zwischen Operationen ist in der Ausgabe ein Leerzeichen anzuhängen.

¹Während Union wird Find gegebenenfalls mehrere Male aufgerufen. Dies sollte in der Gesamtanzahl an traversierten Leveln berücksichtigt sein.

Hilfsroutinen

malloc/free

In dieser Aufgabe stehen euch malloc und free zur Verfügung. Der verfügbare Speicher ist begrenzt, reicht jedoch in jedem Fall für Lösung der jeweiligen Aufgabe. Der dynamisch allozierte Speicher sollte wieder freigegeben werden.

getint

```
1 uint64_t getint(char *str)
```

Die Hilfsroutine getint liest eine Zahl aus dem übergebenen String str und gibt sie als vorzeichenlose 64-Bit Zahl zurück. Es werden so lange Ziffern aus dem String gelesen, bis das erste Zeichen, das nicht eine Ziffer ist, gefunden wird. Die Routine geht davon aus, dass auf die letzte Ziffer entweder ein weiteres Zeichen oder der Null-Terminator folgt.

Zusatzfeature: Obwohl es nicht ganz der Calling-Convention entspricht, dürft ihr davon ausgehen, dass nach dem ausführen von getint der übergebene char* str (bzw. dessen Register) auf das Zeichen hinter der letzten gelesenen Ziffer zeigt.

putint

```
1 char* putint(uint64_t value, char* string)
```

Die Hilfsroutine putint nimmt eine Zahl value und hängt sie Ziffer für Ziffer an den String string an. Das erste geschriebene Zeichen ist string[0]. Der Rückgabewert zeigt auf das Zeichen hinter der letzten geschriebenen Ziffer. Die Routine geht davon aus, dass in string immer genügend Platz vorhanden ist, um das Resultat zu schreiben. Der String wird nicht mit einem terminierenden Null-byte versehen.

Rahmenbedingungen

- $n \in [0, 65535]$, $\text{strlen}(\text{instructionString}) \in [0, 65535]$
- Ihr dürft davon ausgehen, dass instructionString valide ist. Insbesondere enthält instructionString keine ungültigen Zeichen oder Referenzen auf ungültige Indices.
- Der solutionString enthält immer genügend Platz, um die Lösung dort abzuspeichern.
- Ihr dürft Online-Ressourcen verwenden. Wenn ihr kleine Codestücke übernehmt, versteht diese mit einer Quellenangabe im Kommentar. Der Großteil der Aufgabe ist jedoch selbstständig zu lösen.

- Achtet auf die Calling Convention, wenn ihr Hilfsfunktionen aufruft.
 - Sobald die Aufgabe im Aufgabentester als korrekt gelöst markiert ist, habt ihr die Zusatzaufgabe bestanden.
 - Plagiate oder die Nutzung von Compiler-generiertem Code führen zur Aberkennung der Zusatzaufgabe und damit zum nicht-bestehen des Praktikums.
 - Eure Tutoren sind für euch da. Allerdings ist diese Aufgabe ein Teil eurer Prüfungsleistung. Inhaltliche Fragen können dementsprechend nur in sehr begrenztem Maße beantwortet werden.
 - Abgabefrist: Auf Praktikumswebseite zu finden.
-