



Übungen zu Woche 4: Amortisierte Analyse

Dienstag

moodle 🧡 4.1 Amortisierte Analyse. Hier ist eine Funktion $f(n)$, die eine andere Funktion $g()$ aufruft:

```
def f(n):  
    for i in range(n):  
        g()
```

Wir nehmen an, dass der i -te Aufruf von $g()$ eine Laufzeit von $T(i)$ hat, wobei

$$T(i) = \begin{cases} 2i & \text{falls } i \text{ eine Zweierpotenz ist;} \\ 1 & \text{sonst.} \end{cases}$$

Analysiere die amortisierte Laufzeit von $g()$.

- Benutze die Aggregationsmethode.
- Benutze das Buchhalter-Argument (*accounting method*).

4.2 Mengenvereinigungen. Wir betrachten eine abstrakte Datenstruktur, die disjunkte Teilmengen von $\{1, \dots, n\}$ verwaltet (ganz ähnlich zur Union-Find). Anfangs sind die Elemente auf n einzelne Mengen aufgeteilt, also $\{1\}, \{2\}, \dots, \{n\}$. Die Datenstruktur unterstützt die folgenden Operationen:

- Union(A, B): Führe die beiden Mengen A und B zu einer neuen Menge $C = A \cup B$ zusammen und lösche die alten Mengen A und B .
- SameSet(x, y): Gebe *true* zurück, wenn x und y in derselben Menge liegen, ansonsten gebe *false* zurück.

Wir implementieren diese Operationen nun nicht mit *Weighted Quick-Union*, sondern mithilfe einer Variante von *Quick-Find*, die wir *Weighted Quick-Find* nennen. Hierbei speichern wir wie bei *Quick-Find* ein Array $F[1 \dots n]$, das jedem Element eine „Farbe“ zuordnet; eine Zahl $F[i] \in \{1, \dots, n\}$ heißt hierbei, dass Element i die Farbe $F[i]$ erhält. Außerdem wird ein Array $M[1 \dots n]$ aufrecht erhalten, das für jede Farbe $f \in \{1, \dots, n\}$ die Einträge $M[f].list$ und $M[f].length$ enthält. Hier ist $M[f].list$ eine verkettete Liste aller Elemente $i \in \{1, \dots, n\}$ mit $F[i] = f$ und $M[f].length$ ist die Länge der Liste $M[f].list$. Die Union-Operation übernimmt für alle Elemente in der kleineren Menge die Farbe der größeren Menge (bei Gleichstand wird die Farbe der Menge A gewählt). Die SameSet-Operation überprüft, ob die zwei Elemente dieselbe Farbe haben.

Wir werden nun folgendes beobachten: Aus Sicht der worst-case Laufzeit ist die Union-Operation von *Weighted Quick-Find* viel schlechter als *Weighted Quick-Union*, aber die amortisierten Kosten sind asymptotisch gleich.

- 🧡 Schreibe die Datenstruktur mit ihren beiden Operationen als Code oder Pseudocode auf.
- 🧡 Analysiere die *worst-case* Laufzeit der beiden Operationen.
- 🧡 🔑 Zeige, dass die amortisierten Kosten $O(\log n)$ für Union und $O(1)$ für SameSet betragen.
- 🧡 🔑 Zeige ebenfalls, dass jede Sequenz von m Union-Operationen und l SameSet-Operationen die worst-case Laufzeit $O(m \log n + l)$ einhält. (Tipp: ⚠️ Siehe auch die Farbe jedes Elements ein array F wie oft wir)

Donnerstag

moodle 🧠 **4.3 Potenzialfunktion: Verdopplung von Arrays.** Wir betrachten einen Stack, in dem wir nur push und nicht pop erlauben. Der Stack wird durch ein dynamisch wachsendes Array dargestellt ist, das heißt, das Array wird verdoppelt, wenn push aufgerufen wird und das Array voll ist. Sei A_i das Array nach der i -ten Operation. Wir definieren die Potenzialfunktion $\Phi(A_i) = k$, wobei k die aktuelle Anzahl an belegten Elementen im Array ist. Können wir mit dieser Potenzialfunktion zeigen, dass die amortisierten Kosten von push konstant sind? Wenn ja, wie? Wenn nein, warum nicht?

4.4 Dynamische Hashtabelle. Wir wollen nun eine Hashtabelle mit linearem Sondieren so implementieren, dass das verwendete Array mit der Verdopplungsmethode dynamisch wächst. Deine Datenstruktur soll also die folgenden Operationen unterstützen:

- `init()` initialisiert eine neue, leere Hashtabelle.
- `set(k, v)` fügt das Schlüssel-Wert-Paar (k, v) in die Hashtabelle ein.
- `get(k)` liefert den Wert, der mit dem Schlüssel k assoziiert ist, oder `null`, falls dieser nicht in der Hashtabelle enthalten ist.


Entwirf nun eine effiziente Datenstruktur, die nur $O(n)$ Speicher verwendet, wobei n die aktuelle Anzahl der Elemente in der Hashtabelle ist.




- 🧠 Beschreibe deine Datenstruktur in Code oder Pseudocode.
- 🧠 Was ist die worst-case und die amortisierte Laufzeit deiner Implementierung von `set(k, v)`?
- 🧠 🚧 Manchmal ist es möglich, Datenstrukturen zu „deamortisieren“. Das heißt, man erhält dieselben *worst-case* Schranken wie im amortisierten Fall, indem die teure Arbeit auf viele Operationen verteilt wird. Verändere nun deine Datenstruktur so, dass die *worst-case* Laufzeit konstant ist, aber weiterhin nur $O(n)$ Speicherplatz benutzt wird. (Tipp: $\forall n \in \mathbb{N}$ existiert ein $k \in \mathbb{N}$ mit $k \leq n$ und $k \equiv 1 \pmod{2}$)


Weitere Aufgaben und mögliche Projekte

 **4.5 Splay-Bäume.** Gegeben sei ein leerer Splay-Baum.

- Füge die Schlüssel 41, 38, 31, 12, 19, 8 in der angegebenen Reihenfolge ein. Wie sieht der resultierende Splay-Baum aus?
- Wie sieht der Splay-Baum aus, nachdem man Schlüssel 31 entfernt hat?



 **4.6 Spiel-Bäume.** Professor Sheldon schlägt die sogenannten Spiel-Bäume als eine einfachere Variante der Splay-Bäume vor. Hierbei verzichten wir innerhalb der $\text{splay}(x)$ -Methode auf die nervigen *roller-coaster* und *zig-zag* Transformationen, und führen stattdessen nur einfache Rotationen durch—so lange, bis x an der Wurzel landet.


-  Was sind die amortisierten Kosten der Operation $\text{splay}(x)$, wenn wir nur einfache Rotationen benutzen? Analysiere die Kosten mit derselben Potenzialfunktion, die wir für Splay-Bäume verwendet haben.
-  Was sind die gesamten (tatsächlichen) Kosten, wenn man zuerst n Elemente mit den Schlüsseln $1, 2, 3, \dots, n$ der Reihenfolge nach in einen Spiel-Baum einfügt und dann in der gleichen Reihenfolge sucht?
-  Professor Sheldon behauptet, dass das Einfügen, Suchen und Löschen in Spiel-Bäumen amortisierte Kosten von $O(\log n)$ hat. „Du musst nur eine raffiniertere Potenzialfunktion benutzen“, sagt er. Liegt er damit richtig?

 **4.7 Queues mit zwei Stacks.**


Wir erinnern uns, dass ein Stack S eine Datenstruktur ist, die die Operationen $S.\text{PUSH}(x)$, $S.\text{POP}()$ und $S.\text{ISEMPTY}()$ unterstützt. Wir nehmen an, dass diese Operationen in konstanter *worst-case* Zeit implementiert worden sind.

Du sollst für diese Aufgabe eine Queue Q mithilfe von zwei Stacks S_1, S_2 implementieren. Dabei darfst du nur $O(1)$ zusätzlichen Speicher benutzen. Der *einzig*e Zugriff auf die Stacks erfolgt als *black-box* über die Standardoperationen PUSH , POP und ISEMPTY .

-  **Beschreibe mit Pseudocode** eine solche Queue-Implementierung, wobei die amortisierte *worst-case* Laufzeit für jede $Q.\text{ENQUEUE}(x)$, $Q.\text{DEQUEUE}()$ und $Q.\text{ISEMPTY}()$ Operation konstant sein muss.
-  **Beweise mithilfe der Potenzialmethode**, dass die amortisierten Kosten deiner Implementierung tatsächlich konstant sind.

 **4.8 Implementierung von dynamischen Tabellen.** Implementiere deine eigene dynamische Tabelle für Integer-Werte, ohne dabei built-in Methoden zu verwenden. Deine dynamische Tabelle soll folgende Operationen beherrschen:


- Einfügen von Elementen,
- Löschen von Elementen,
- Ausgabe enthaltener Elemente,
- Ausgabe der Größe der Tabelle.

 **4.9 Deamortisierung für Stacks mit dynamischen Arrays.** Manchmal ist es möglich, Datenstrukturen zu „deamortisieren“. Das heißt, man erhält dieselben *worst-case* Schranken wie im amortisierten Fall, indem die teure Arbeit auf viele Operationen verteilt wird.

- Wir betrachten einen Stack, der nur push unterstützen soll. Wie kann man push so implementieren, dass die *worst-case* Laufzeit konstant ist, aber weiterhin nur $O(n)$ Speicherplatz benutzt wird, wo n die aktuelle Anzahl der gespeicherten Elemente ist?

(Tipp: verwende alle Funktionen die dir erlaubt sind, aber vermeide unnötige Komplexität)

b) Jetzt mach dasselbe für Stacks, die push *und* pop unterstützen.

 **4.10 Puzzle der Woche: Prinzessinnen.** Stelle dir vor, du bist ein junger Prinz aus dem fernen Lande Algo und der König des benachbarten Landes Logik hat 3 Töchter. Die Älteste erzählt immer die Wahrheit, die Jüngste lügt immer und die Mittlere lügt und sagt die Wahrheit, wie es ihr gefällt.

Du willst entweder die älteste oder die jüngste Tochter heiraten, da immer lügen genauso gut ist, wie immer die Wahrheit zu sagen. Nur die mittlere Tochter möchtest du nicht heiraten. Allerdings sehen alle Töchter gleich aus, sodass du sie nicht unterscheiden kannst.

Der König ist ein hinterhältiger Mann und erlaubt dir, genau *eine* Frage an genau *eine* der drei Töchter zu stellen. Diese Frage soll mit „Ja“ oder „Nein“ zu beantworten sein. Danach musst du dich entscheiden, welche der drei Prinzessinnen du heiraten möchtest.

Welche Frage solltest du stellen und dann welche der Töchter aussuchen?