Technische Universität München
Fakultät für Informatik, I-14
Prof. Harald Räcke
Robin Münk

SoSe 2025
Übungsblatt 9
20. Juni 2025

Algorithmen und Datenstrukturen

Aufgabe 1 QuickSort - Median of Medians

Der Median einer Liste an Zahlen ist der Wert, welcher in der sortierten Liste genau in der Mitte stehen würde und somit die Hälfte der größeren Zahlen von der Hälfte der kleineren Zahlen trennt. Es gibt somit gleich viele Werte in der Liste die größer sind, als der Median, wie Zahlen, die kleiner sind als der Median.

Der Median einer Liste kann zum Beispiel mit dem BFPRT-Algorithmus gefunden werden. Für die Laufzeit dieses rekursiven Algorithmus gilt folgende Rekurrenz:

$$T(n) \leq \begin{cases} c & n \leq 1 \\ T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + cn & \text{sonst} \end{cases}$$

für ein $c \geq 1$.

a) Lösen Sie die Rekurrenz, indem Sie mit vollständiger Induktion beweisen, dass für alle $n \geq 1$ gilt $T(n) \leq 10 \cdot cn$.

b) Stellen Sie eine Rekursionsgleichung für die Laufzeit von QuickSort auf, wenn als Subroutine für die Bestimmung des Pivot-Elements in jedem Schritt der BFPRT-Algorithmus verwendet wird.

c) Lösen Sie die Rekurrenz aus b) und geben Sie einen geschlossenen Ausdruck für eine obere Schranke an die Laufzeit dieser Variante von QuickSort an.
Vergleichen Sie das Ergebnis mit der Laufzeit von MergeSort und der asymptotischen worst-case Laufzeit von QuickSort aus der Vorlesung.

Beispiellösung:

a)
- Induktionsanfang $n = 1$:
  $T(1) \leq c \leq 10 \cdot c \cdot 1 = 10c \rightarrow$ Stimmt, da $c \geq 1$.
- Induktionsschritt $0, ..., n-1 \rightarrow n$
  Angenommen, die Aussage ist wahr für $n' \in \{1, ..., n-1\}$ und $n > 1$. Dann:

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + cn \\ &\leq 10c\frac{n}{5} + 10c\frac{7n}{10} + cn \\ &= cn\left(10 \cdot \frac{1}{5} + 10 \cdot \frac{7}{10} + 1\right) \\ &= cn\,(2+7+1) \\ &= 10 \cdot cn \end{aligned}$$

b) Da der BFPRT-Algorithmus das Median-Element findet, wird bei jeder Iteration von QuickSort die Liste in zwei gleich große Hälften geteilt. Zusammen mit dem Ergebnis aus a) ergibt sich also

$$T(n) = \begin{cases} O(1) & n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + 10cn + O(n) & \text{sonst} \end{cases}$$

c) Wir lösen zunächst die O-Notation auf. Für die Rekurrenz ergibt sich damit

$$T(n) \leq \begin{cases} d & n \leq 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + dn & \text{sonst} \end{cases}$$

mit einer hinreichend großen Konstante d. Es ergibt sich die selbe Rekurrenz, wie für MergeSort, welche aus der Vorlesung bereits bekannt ist.

Zur Lösung mit dem Master-Theorem gilt also: $a = 2, b = 2, f(n) = dn$ und somit $\log_b a = \log_2 2 = 1$. Da außerdem $f(n) = dn \in \Theta(n^1 \cdot \log^0 n) = \Theta(n^{\log_b a} \cdot \log^k n)$, ist Fall 2 des Master-Theorems für $k = 0$ anwendbar.
Daraus folgt als Schranke für die Laufzeit: $T(n) \in \Theta(n^{\log_b a} \cdot \log^{k+1} n) = \Theta(n \log n)$.

Bedeutung: Das zeigt also, wenn wir den BFPRT-Algorithmus verwenden, um den Median einer Liste in $O(n)$ Zeit finden, ist die asymptotische worst-case Laufzeit von QuickSort in $\Theta(n \log n)$. Das ist eine deutliche asymptotische Verbesserung zu $O\left(n^2\right)$ bei der Variante mit trivialer Pivot-Auswahl!
Meistens haben in der Praxis allerdings trotzdem Implementierungen mit zufälliger Pivot-Wahl in $O(1)$ eine schnellere Laufzeit, da der extra-Aufwand bei der Selektion große Konstanten in der O-Notation versteckt.

Aufgabe 2 Stack und Queue

a) Gegeben sei ein Stack S:

$$\leftrightarrow (5, 8, 2, 3, 9)$$

Dabei sind folgende Operationen auf dem Stack definiert:

- S.`pop()`: Liefert das oberste Objekt und entfernt es vom Stack S.
- S.`push(`x`)`: Legt das Objekt x oben auf den Stack S.
- S.`top()`: Liefert das oberste Objekt ohne es zu entfernen.
- S.`isEmpty()`: Gibt zurück, ob der Stack S leer ist.

Geben Sie jeweils den Zustand von S und die Rückgabe an, wenn folgende Operationen ausgeführt werden:

(i) S.`pop()`

(ii) S.`push(8)`

(iii) S.`top()`

(iv) S.`push(1)`

(v) S.`pop()`

b) Gegeben sei eine Queue Q:

$$\rightarrow (0,9,1,3,4) \rightarrow$$

Dabei sind folgende Operationen auf der Queue definiert:

- Q`.enqueue(`x`)` Fügt das Objekt x and das Ende der Queue Q.
- Q`.dequeue()` Liefert und entfernt das am längsten gespeicherte Objekt aus Q.
- Q`.isEmpty()` Gibt zurück, ob die Queue Q leer ist.

Geben Sie jeweils den Zustand von Q und die Rückgabe an, wenn folgende Operationen ausgeführt werden:

(i) Q`.dequeue()`
(ii) Q`.enqueue(7)`
(iii) Q`.enqueue(5)`
(iv) Q`.dequeue()`
(v) Q`.isEmpty()`

Beispiellösung:

a)

$$\leftrightarrow (5,8,2,3,9)$$

(i) S.pop() = 5

$$\leftrightarrow (8,2,3,9)$$

(ii) S.push(8)

$$\leftrightarrow (8,8,2,3,9)$$

(iii) S.top() = 8

$$\leftrightarrow (8,8,2,3,9)$$

(iv) S.push(1)

$$\leftrightarrow (1,8,8,2,3,9)$$

(v) S.pop() = 1

$$\leftrightarrow (8,8,2,3,9)$$

b)

$$\rightarrow (0,9,1,3,4) \rightarrow$$

(i) Q.dequeue() = 4

$$\rightarrow (0,9,1,3) \rightarrow$$

(ii) Q.enqueue(7)

$$\rightarrow (7,0,9,1,3) \rightarrow$$

(iii) Q.enqueue(5)

$$\rightarrow (5,7,0,9,1,3) \rightarrow$$

(iv) Q.dequeue() = 3

$$\rightarrow (5,7,0,9,1) \rightarrow$$

(v) Q.isEmpty() = false

$$\rightarrow (5,7,0,9,1) \rightarrow$$

Aufgabe 3 Queue aus Stacks

a) Realisieren Sie eine Queue mit Hilfe zweier Stacks. Nehmen Sie an, sie haben zwei Stacks mit den zugehörigen Methoden *push*, *pop* und *isEmpty* zur Verfügung.
Implementieren Sie folgende Methoden. Sie können das Testing Framework verwenden.

- enqueue(x)

- dequeue()
- isEmpty()

Tipp: Nutzen Sie Stack 1 zum Einfügen und Stack 2 zum Entfernen von Elementen.

b) Nehmen Sie an, es befinden sich n Objekte in der Queue. Geben Sie die best-case und die worst-case Laufzeit der Operationen *enqueue* und $dequeue$ an.

c) Zeigen Sie, dass die drei Operationen *enqueue*, $dequeue$ und $isEmpty$ konstante amortisierte Laufzeit haben.

Beispiellösung:

a) Der Code ist in der Datei `blatt9solution.cpp` zu finden.

b) Enqueue:

(i) best-case: konstant $\rightarrow O(1)$

(ii) worst-case: konstant $\rightarrow O(1)$

Dequeue:

(i) best-case: Stack 2 ist nicht leer, konstant $\rightarrow O(1)$

(ii) worst-case Stack 2 ist leer, Elemente von Stack 1 müssen auf Stack 2 umgestapelt werden. Laufzeit hängt von der Anzahl der Element in Stack 1 ab. Im schlimmsten Fall also n. $\rightarrow O(n)$

c) Angenommen, wir beginnen mit einer leeren Queue und es werden m Operationen auf der Queue durchgeführt (enqueue oder dequeue). Wir möchten die Laufzeit für diese m Operationen analysieren.

Man kann es sich wie folgt vorstellen: Anstatt nur eine Münze (konstante Kosten) für jede enqueue Operation zu bezahlen, legen wir eine Münze zusätzlich in eine "Spardose". D.h. wir bezahlen bei jeder enqueue Operation die Kosten für das Umstapeln des hinzugefügten Elements im Voraus. Wenn wir bei einer dequeue Operation Elemente vom Stack 1 nach Stack 2 umstapeln, zahlen wir mit den Münzen aus der Spardose. Wir haben immer genug Münzen in der Spardose, da wir für jedes Element in Stack 1 eine Münze in die Spardose eingezahlt haben. D.h. bei einer dequeue Operation sind die Kosten für das Umstapeln schon gedeckt.

Es lassen sich also $O(1)$ amortisierte Kosten für beide Operationen enqueue und dequeue erreichen. Dann ist die Laufzeit von m Queue Operationen in $O(m)$.

Aufgabe 4 Stack - Rekursive Sortierung

a) Gegeben ist ein Stack S. Implementieren Sie eine Funktion **stackSort(S)**, die den Stack S *rekursiv* aufsteigend sortiert. Implementieren Sie dafür eine *rekursive* Hilfsfunktion **sortedInsert(S, val)**, die den Wert *val* in den bereits sortierten Stack S an die richtige Stelle einfügt. Sie können das Testing Framework verwenden.

Der Stack soll dabei nur über die bekannten Stackfunktionen pop, push, top, length und isEmpty benutzt werden.

Beispiel:
Stack vor der Sortierung: $\leftrightarrow$ [5, 3, 2, 6, 7]
Stack nach der Sortierung: $\leftrightarrow$ [2, 3, 5, 6, 7]

b) Analysieren Sie die Laufzeit der Funktion **sortedInsert**. Leiten Sie daraus die Laufzeit von **stackSort** ab.

Beispiellösung:

a) Der Code ist in der Datei `blatt9solution.cpp` zu finden.

b) Sei n die Anzahl der Elemente im Stack. Die Kosten der Stackmethoden sind konstant.

Laufzeit sortedInsert: $O(n)$
Außer des rekursiven Aufrufs sind die Kosten der anderen Anweisungen konstant. SortedInsert kann sich maximal n mal rekursiv aufrufen (bei jedem Aufruf reduziert sich die Anzahl der Elemente im Stack). Also sind die Kosten in $O(n)$

Laufzeit stackSort: $O(n^2)$
Die Kosten des pop-Aufrufs sind konstant und die Kosten von sortedInsert sind in $O(n)$. D.h., abgesehen von dem rekursiven Aufruf sind die Kosten in $O(n)$. Genauso wie sortedInsert, kann sich stackSort maximal n mal rekursiv aufrufen. Jedes Mal wird die Anzahl der Elemente um eins reduziert. Wir müssen die Kosten der Rekursionsaufrufe addieren, um die gesamten Kosten zu berechen. Die Kosten des ersten Aufrufs sind in $O(n)$, des zweiten in $O(n-1)$ (Stackgröße wird um eins reduziert), des dritten in $O(n-2)$, usw. Da $\sum_{i=1}^{n} i = n(n+1)/2$, sind die gesamten Kosten in $O(n^2)$.