Karlsruher Institut für Technologie

Institut für Theoretische Informatik
T.T.-Prof. Dr. Thomas Bläsius
https://scale.iti.kit.edu

# Übungsblatt 03

## Algorithmen I - Sommersemester 2024

**Abgabe im ILIAS bis 10.05.2024, 18:00 Uhr**

Die Abgabe erfolgt alleine oder zu zweit als eine PDF-Datei über das Übungsmodul in der Gruppe eures Tutoriums im ILIAS. Beschriftet die Abgabe deutlich mit Matrikelnummer(n) und Name(n). Bei Abgaben zu zweit reicht es, wenn eine Person die Abgabe im ILIAS hochlädt.

- Achtet bei handschriftlichen Abgaben auf Lesbarkeit.
- Achtet darauf, ob Algorithmen in Worten oder Pseudocode beschrieben werden sollen.
- Es werden immer asymptotisch möglichst effiziente Algorithmen erwartet, wenn nicht anders angegeben.
- Wenn Korrektheits- oder Laufzeitanalysen gefordert sind, behandelt diese separat von der Algorithmenbeschreibung.

**Gesamtpunkte**: 20

## Aufgabe 1 - The odds are stacked against you (7 Punkte)

In dieser Aufgabe soll die Datenstruktur **Stack** mithilfe von Listen und Arrays umgesetzt werden.
Ein Stack unterstützt hierbei die folgenden Operationen:

- `push`(x): Legt das Element x oben auf den Stack.
- `pop`(x): Entfernt das oberste Element vom Stack und gibt es zurück.

Viele Stacks unterstützen außerdem die Operation `peek`. Da Dr. Meta diese allerdings sehr langweilig findet, möchte er, dass dein Stack stattdessen die folgende (laut ihm spannendere) Operation umsetzt.

- `spy`(): Gibt das aufgerundet mittlere Element des Stacks zurück ohne es zu entfernen.

In dieser Aufgabe sollst du auf Zeiger- bzw. Arrayeintragebene argumentieren und nicht die angebotenen Operationen der Datenstrukturen aufrufen.

a) Beschreibe, wie ein solcher Stack mit Hilfe einer **doppelt** verketteten Liste (Implementierung von Folie 3, ohne Dummy-Knoten) effizient modelliert werden kann. Gib an, welche zusätzlichen Daten (wie z. B. Zeiger oder Variablen) benötigt werden und wie `push`(x), `pop`(x) und `spy`() funktionieren. Gib an, welche Laufzeit deine Operationen benötigen und begründe warum. (3 Punkte)

b) Begründe, warum bei Verwendung, von **einfach** verketteten Listen mindestens eine der drei Operationen mehr als konstante Laufzeit benötigt. (1 Punkt)

c) Beschreibe, wie ein solcher Stack mit Hilfe eines dynamischen Arrays effizient modelliert werden kann. Gib an, welche zusätzlichen Daten (wie z. B. Zeiger oder Variablen) benötigt werden und wie `push`(x), `pop`(x) und `spy()` funktionieren. Gib an, welche amortisierte Laufzeit deine Operationen benötigen und begründe warum.

Hinweis: Du kannst hier die amortisierten Laufzeiten aus der Vorlesung verwenden und musst selbst keine amortisierte Analyse durchführen. (3 Punkte)

## Lösung 1

a) Wir fügen hier die neuen Elemente am Anfang der Liste ein und speichern die Elemente des Stacks also in gewisser Weise in der gespiegelten Reihenfolge.

Wir speichern uns zusätzlich zu unserer Liste, wie in der Vorlesung, noch einen `head`-Zeiger, welcher auf den ersten Knoten zeigt. Außerdem speichern wir uns einen `mid`-Zeiger, der immer auf das aufgerundet mittlere Element zeigt und in `size` die Anzahl der Elemente im Stack. Um diese Zeiger und Werte zu speichern, wird konstanter zusätzlicher Speicherbedarf benötigt.

Bei `push` wird ein neuer Knoten angelegt und vor den aktuellen `head`-Knoten gehangen. Hierfür zeigt der `next`-Zeiger des neuen Knotens auf den aktuellen `head`-Knoten und der `prev`-Zeiger des aktuellen `head`-Knotens auf den neuen Knoten. Der `head`-Zeiger zeigt danach auf den neuen Knoten. Da hierfür nur konstant viele Zeiger verändert werden und wir auf den `head`-Knoten in konstanter Zeit zugreifen können, braucht diese Operation nur $\Theta(1)$ Zeit.

Bei `pop` wird der Wert des `head`-Knotens ausgegeben und der `head`-Pointer auf den Nachfolger des aktuellen `head`-Knotens gesetzt. Außerdem muss der `prev`-Zeiger, des neuen `head`-Knotens gelöscht werden. Auch hier müssen wir nur konstant viele Zeiger umsetzen und können auf die benötigten Elemente in konstanter Zeit zugreifen. Daher braucht auch diese Operation nur $\Theta(1)$ Zeit.

Bei beiden Operationen passen wir `size` der Operation entsprechend an. Falls `size` von einer geraden auf eine ungerade Zahl erhöht wird oder von einer ungeraden auf eine gerade Zahl abgesenkt wird, verschieben wir `mid` in die entsprechende Richtung. Da Addition nur konstante Zeit benötigt und wir durch `next` und `prev` in konstanter Zeit zum nächsten Knoten navigieren können, benötigt diese Operation nur konstante Zeit.

Bei `spy` wird der Wert des `mid`-Knotens ausgegeben. Hier müssen wir keine Zeiger umsetzen und können auf die benötigten Elemente in konstanter Zeit zugreifen. Daher braucht auch diese Operation nur $\Theta(1)$ Zeit.

b) Durch die Verwendung von `push` und `pop` kann sich die Mitte des Stacks so wohl nach vorne als auch nach hinten verschieben. Für `spy` gibt es generell zwei Optionen: Speichere die Mitte ab, dann kann man allerdings nur in die Richtung der `next`-Zeiger verschieben, ohne die gesamte Liste durchlaufen zu müssen. Oder alternativ: Bestimme `mid` bei jedem Aufruf von `spy` neu. Dies benötigt offensichtlicherweise lineare Zeit.

c) Hier fügen wir neue Elemente stets an das Ende des Arrays und können so `pushBack` und `popBack` aus der Vorlesung nutzen.

Wir speichern uns in einem `free`-Zeiger die erste freie Stelle im Array. Erneut speichern wir uns einen `mid`-Zeiger, der immer auf das aufgerundet mittlere Element zeigt und in `size` die Anzahl der Elemente im Stack. Um diese Zeiger zu speichern, wird konstanter zusätzlicher Speicherbedarf benötigt.

Bei `push` wird das Element and die Stelle des `free`-Zeiger gespeichert und dieser um eins nach rechts verschoben. Wenn das Array bereits voll ist, vergrößern wir dieses wie in der Vorlesung gezeigt. Es kann analog zur Vorlesung gezeigt werden, dass die amortisierte Laufzeit in $\Theta(1)$ liegt.

Bei `pop` wird der `free`-Zeiger um eins nach links verschoben und anschließend der Eintrag an der neuen Position des Zeiger zurückgegeben. Optional können wir das Array verkleinern, falls der Füllstand zu niedrig ist. Es kann analog zur Vorlesung gezeigt werden, dass die amortisierte Laufzeit in $\Theta(1)$ liegt.

Wir passen `size` und `mid` wie oben an. Hier können wir den Zeiger durch merken des Indexes umsetzen und so mittels Addition in konstanter Zeit anpassen.

Bei `spy` wird der Eintrag an der Position des `mid`-Zeigers ausgegeben. Hier wir müssen keine Zeiger umsetzen und können auf die benötigten Elemente in konstanter Zeit zugreifen. Daher braucht auch diese Operation nur $\Theta(1)$ Zeit.

Beachte, dass `mid` nicht bei jedem Aufruf von `push` oder `pop` angepasst wird und daher in irgendeiner Form gespeichert werden muss, wann das Verschieben erfolgt. Oben wurde eine Möglichkeit umgesetzt. Es gibt viele weitere korrekte Ansätze.

## Aufgabe 2 - Ich versteh nur Bahnhof (6 Punkte)

Wir betrachten die folgenden zwei Algorithmen REVERSE und BAHNSTREIK. Beachte, dass wir dabei häufig über Mengen iterieren, bei denen eine implizite Ordnung angenommen wird.

REVERSE($A : [\mathbb{N}_0, n]$, start : $\mathbb{N}_0$, end : $\mathbb{N}_0$) kehrt die Reihenfolge der Elemente in $A$ im Intervall $[\text{start}, \text{end}]$ um. Die Grenzen werden hierbei mitgetauscht. Dafür geht der Algorithmus von außen nach innen über das Array und vertauscht wiederholt die äußeren beiden Elemente.

```
1:  BAHNSTREIK(A : [ℕ_0, n])
2:      for Verhandlungsrunde ∈ {n, ..., 2} do   \\Absteigend
3:          Vermittlungsvorschlag := 0
4:          for Forderung ∈ {0, ..., Verhandlungsrunde − 1} do
5:              if A[Forderung] > A[Vermittlungsvorschlag] then
6:                  Vermittlungsvorschlag = Forderung
7:              end
8:          end
9:          REVERSE(A, 0, Vermittlungsvorschlag)
10:         REVERSE(A, 0, Verhandlungsrunde − 1)
11:     end
```

a) Gegeben sei das Array: $A = [7, 4, 33, 1, 0, 5, 12, 13, 2, 11]$
Führe jeden der zwei Algorithmen auf $A$ aus und gib den resultierenden Inhalt von $A$ an. Verwende für REVERSE die Parameter start $= 2$, end $= 7$. (2 Punkte)

b) Gebe für REVERSE Pseudcode an und beschreibe BAHNSTREIK in Worten. Bestimme und begründe außerdem deren asymptotische Laufzeiten in Abhängigkeit aller Eingabeparameter. (4 Punkte)

## Lösung 2

a)
- REVERSE: $A = [7, 4, 13, 12, 5, 0, 1, 33, 2, 11]$
- BAHNSTREIK: $A = [0, 1, 2, 4, 5, 7, 11, 12, 13, 33]$

b) Es ergibt sich der folgende Pseudocode:

```
1:  REVERSE(A : [ℕ_0, n], start : ℕ_0, end : ℕ_0)
2:      tmp := 0
3:      while start < end do
4:          tmp = A[start]
5:          A[start] = A[end]
6:          A[end] = tmp
7:          start = start + 1
8:          end = end − 1
9:      end
```

Der Algorithmus iteriert über $\frac{\text{end}-\text{start}}{2}$ Elemente und führt jeweils eine Vertauschungsoperation aus. Da diese in konstanter Zeit erfolgt, ist die Gesamtlaufzeit $\Theta(\text{end} - \text{start})$.

BAHNSTREIK sortiert das Array, indem es das Array in einen unsortierten und einen sortierten Teil aufteilt. In jedem Schleifendurchlauf wird nun das maximale Element im noch unsortierten Teils gefunden und an den Anfang des sortierten

Teils bewegt. Dafür wird es durch REVERSE erst an den Anfang des Arrays getauscht und anschließend wird der unsortierte Teil umgedreht.

In jedem Schleifendurchlauf erfolgen drei Operationen. Zuerst wird das Maximum im unsortierten Bereich gefunden, anschließend werden zwei Teilabschnitte des Arrays umgedreht. Hierbei ist der zweite Abschnitt stets größer und dominiert daher die Laufzeit. Sowohl das zweite Drehen, als auch das Finden des Maximums ist linear in der Größe des unsortierten Bereichs. Somit ist die Laufzeit:

$$\begin{aligned}\sum_{i=2}^{n} \Theta(i) &= \Theta(\frac{n \cdot (n+1)}{2}) \\ &= \Theta(n^2)\end{aligned}$$

## Aufgabe 3 - Knapp daneben ist ~~auch vorbei~~ nicht sofort vorbei (7 Punkte)

Nachdem sich die Klon-Maschine, wie von dir auf dem letzten Blatt bewiesen, als höchst erfolgreich herausgestellt hat, möchte sich Dr. Meta einen Überblick über seine Biber verschaffen. Dafür hat er die Biber eindeutig durchnummeriert. Für die nächste Stufe seines Geheimplans möchte er bestimmte Biber in der Menge der Biber finden können. Da der Biber, der die Nummerierung durchgeführt hat, einige Nummern übersprungen hat, lässt Dr. Meta alle Biber sich in einer Reihe nach ihrer Nummer sortiert aufstellen. Hier möchte er mittels einem von ihm eigens entwickelten und hoch innovativem Algorithmus namens binärer Suche die gesuchten Biber ausfindig machen. Da er in weiser Voraussicht den Boden mit Indizes beschriftet hat, kann er bei gegebenem Index in konstanter Zeit auf den Biber an dieser Position zugreifen.

Nach einigen Testläufen stellt Dr. Meta allerdings fest, dass die Biber einen Fehler bei ihrer Aufstellung gemacht haben: Jeder Biber steht höchstens ein Feld entfernt von der Position, die er in einer korrekt sortierten Folge hätte, allerdings nicht zwingend an dieser. Somit sind die Biber nur *beinahe sortiert*.

a) Zeige, dass die aus der Vorlesung bekannte binäre Suche auf beinahe sortierten Aufstellungen falsche Rückgaben liefern kann. Führe dafür die klassische binäre Suche (Folie 14) auf der folgenden, als Array repräsentierten, Aufstellung von Bibern aus. Suche dabei die Zahl 7. Gebe hierbei die Werte von `beg`, `end` und `mid` in jedem Aufruf an. Gebe außerdem den Rückgabewert an. (1.5 Punkte)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 7 | 2 | 9 | 12 | 10 | 21 | 20 | 42 |

Das Array mit Indizes.

Da die bisher bekannte binäre Suche in diesem Szenario nicht immer korrekt ist, schlägst du Dr. Meta vor, den Algorithmus leicht anzupassen und das Problem so zu lösen. Dr.

Meta ist zwar nicht völlig von deiner Idee überzeugt, beauftragt dich aber mit der Ausarbeitung des modifizierten Algorithmus.

b) Du überlegst dir zuerst, an welchen Indizes ein Biber in einer beinahe sortierten Aufstellung stehen kann. Betrachte hierfür einen Biber, der in der sortierten Aufstellung an Stelle $i$ stehen würde. Wo kann dieser in einer beinahe sortierten Aufstellung stehen? (1 Punkt)

c) Modifiziere den Algorithmus der binären Suche so, dass er die Position des Bibers mit der gegebenen Nummer in einer beinahe sortierten Aufstellung, in welcher der Biber vorkommt, in logarithmischer Zeit bestimmt. Beschreibe deinen Algorithmus in Worten und begründe sowohl seine Laufzeit als auch seine Korrektheit. (3.5 Punkte).

Hinweis: Stelle sicher, dass die binäre Suche das eigentliche mittlere Element auch wirklich betrachtet.

d) Dr. Meta gefällt zwar deine Lösung sehr, um aber spätere Verwirrung zu vermeiden möchte er das zugrunde liegende Problem beheben. Erkläre kurz, wie du in linearer Zeit eine beinahe sortierte Aufstellung sortieren kannst. (1 Punkt)

## Lösung 3

a) Beim initialen Aufruf gilt: `beg` $= 0$, `end` $= 8$ und `mid` $= 4$.
In zweiten Aufruf gilt: `beg` $= 0$, `end` $= 4$ und `mid` $= 2$.
In dritten Aufruf gilt: `beg` $= 3$, `end` $= 4$ und `mid` $= 3$.
In vierten Aufruf gilt: `beg` $= 3$, `end` $= 3$ und `mid` $= 3$.
Wir geben also 3 zurück, obwohl 1 korrekt wäre.

b) Der Biber kann an der selben Position, eins links oder eins rechts stehen. Somit sind die Positionen $i - 1$, $i$ und $i + 1$ möglich.

c) Wir gehen analog zur Vorlesung vor und berechnen in jedem Schrit wie bisher `mid`. Nun betrachten wir die Biber an den Positionen `mid` - 1, `mid` und `mid` + 1. Wenn einer der drei Biber der gesuchte Biber ist, sind wir fertig. Der rekursive Aufruf und die Entscheidung in welchem Bereich wir weiter suchen erfolgen wie bisher.

Für die Korrektheit ist zu zeigen, dass der gesuchte Biber in der beinahe sortierten Aufstellung genau dann links von Position `mid` steht, wenn er es auch in der sortierten Aufstellung tut. Daraus folgt direkt die analoge Aussage für die rechte Hälfte. Außerdem können wir den Fall, dass der gesuchte Biber einer der drei betrachteten Biber ist, direkt ausschließen, da wir in diesem Fall trivialerweise korrekt abbrechen.
Sei der gesuchte Biber also in der linken Hälfte der beinahe sortierten Aufstellung. Dann liegt zwischen dem Biber an Position `mid` und dem gesuchten Biber mindestens eine freie Position. Wir betrachten den worst case mit einer freien Position und bezeichnen diese mit $i$. In der sortierten Folge kann der gesuchte Biber an

Position $i-2$, $i-1$ oder $i$ und der Biber an Position `mid` kann an Position $i$, $i+1$ oder $i+2$ stehen. Da wir eindeutige Nummern vergeben haben, können nicht beide an Position $i$ stehen. Somit steht der gesuchte Biber auch in der sortierten Folge vor `mid` und ist also in der linken Hälfte.
Sei der gesuchte Biber also in der linken Hälfte der sortierten Aufstellung. Wenn er nun nicht einer der betrachteten Biber ist, dann liegt er auch in der linken Hälfte der beinahe sortierten Aufstellung, da er ansonsten mehr als eine Position von seinem sortierten Platz entfernt wäre.

Da wir im Worst Case dieselben Teilbereiche wie eine klassische binäre Suche absuchen und nur konstant viele Biber zusätzlich überprüfen, bleibt die Laufzeit logarithmisch.

d) Wir können die Aufstellung sortieren, indem wir einmal über diese iterieren und die Nummer jedes Bibers mit der Nummer des auf ihn folgenden Bibers vergleichen. Wenn nötig vertauschen wir die beiden Biber nach dem Vergleichen.

Skizze der Korrektheit (nicht gefordert):
Ein Biber, der richtig steht, wird durch diese Vergleiche nicht bewegt, da alle Biber links von ihm kleinere und alle rechts von ihm größere Nummern haben. Um von der einen zur anderen Seite zu wechseln müsste ein Biber mehr als eine Position bewegt werden.

Ein Biber, der ein Feld zu weit rechts steht, hat entweder mit dem Biber, der eigentlich direkt rechts von ihm steht, den Platz getauscht oder diesen ebenfalls nach rechts verschoben. Im ersten Fall wird er durch den Algorithmus mit diesem verglichen und zurück getauscht. Der zweite Fall kann nicht auftreten. Angenommen dieser Fall träte ein, dann setzen sich die Verschiebungen wie eine Kette bis ans rechte Ende der Aufstellung fort. Der rechteste Biber hat dann allerdings keinen Platz mehr in der Aufstellung.

Für Biber, die ein Feld zu weit links stehen folgt analoges.

## Aufgabe 4 - Zum Knobeln (0 Punkte + 2 Bonus)

Gegeben seien die Operationen `pushFront`(x : Element), `popFront`(), `pushBack`(x : Element), `popBack`(), `insertAfter`(k : Knoten, x : Element), `remove`(k : Knoten), `concatenate`(l: Liste, m : Liste)[1] und `splice`(a : Knoten, b : Knoten, c : Knoten) auf doppelt verketteten Listen mit Dummy-Headern.

a*) Mit welcher dieser Operationen kannst du alle anderen in konstanter Zeit umsetzen? (0.5 Punkte)

b*) Beschreibe, wie die anderen Operationen mit deiner in a*) gewähltern Operation umgesetzt werden können. (1.5 Punkte)

[1] `concatenate`(l: Liste, m : Liste) hängt alle Elemente der Liste m hinter das letzte Element der Liste l. Die Reihenfolge der Elemente in jeder Liste bleibt erhalten.

## Lösung 4

a) Mit `splice` können die anderen Operationen umgesetzt werden.

b)
- `pushFront`(x : Element): Erstelle eine Liste, die nur einen Knoten mit Wert x enthält und nutze `splice`, um diesen Knoten hinter den Dummy zu verschieben.
- `popFront`(): Verschiebe mit `splice` den ersten Knoten in eine leere Müll-Liste.
- `pushBack`(x : Element): Analog zu `pushFront`, navigiere über `prev` zum letzten Knoten (füge nach diesem ein).
- `popBack`(): Analog zu `popFront`, navigiere über `prev` zum letzten Knoten.
- `insertAfter`(k : Knoten, x : Element): Erstelle eine Liste, die nur einen Knoten mit Wert x enthält und nutze `splice`, um diesen Knoten hinter den Knoten k zu verschieben.
- `remove`(k : Knoten): Verschiebe mit `splice` den Knoten in eine leere Müll-Liste.
- `concatenate`(l: Liste, m : Liste): Nutze `splice`, um die Liste zwischen dem ersten und letzten Knoten von m hinter das letzte Element von l zu verschieben.