Algorithmen und Datenstrukturen
Blatt 3

Präsentation am 27.–29. November 2019

Jede Teilaufgabe (a, b, ...) zählt als ein einzelnes Kreuzchen.

Übung 1.
Betrachten Sie den folgenden Algorithmus für SelectionSort, der eine Liste von Zahlen absteigend sortiert:

```
SelectionSort(list l of integers)
1  list r := empty list
2  while l is not empty
3       integer min := l.head
4       for e ∈ l
5            if min > e
6                 min := e
7       Insert min at the beginning of r
8       Remove min from l
9  return r
```

(a) Beweisen Sie die Korrektheit von SelectionSort.

(b) Geben Sie eine Laufzeitabschätzung für SelectionSort an. Ist die Laufzeit von SelectionSort für vergleichsbasiertes Sortieren asymptotisch optimal? Was ist der best-case von SelectionSort?

Lösung 1.

(a) Sei n die Anzahl der Elemente in l. Weil in jedem Schleifendurchlauf ein Element aus l entfernt wird (Zeile 8), wird die Schleife insgesamt n Mal durchlaufen.

Wir zeigen die Korrektheit von SelectionSort mittels folgender Schleifeninvariante: $\forall i \in \{1...n\}$:
r ist absteigend sortiert und enthält die kleinsten $i-1$ Elemente aus der Originalliste l, l enthält $n-i+1$ Elemente der Originalliste; alle Elemente sind größer oder gleich den Elementen aus r.

Initialisierung: Die Schleifeninvariante gilt zu Beginn des ersten Schleifendurchlaufs, $i = 1$. Dies ist offensichtlich, da r leer ist, also die kleinsten 0 Elemente von l in absteigender Reihenfolge enthält, und l alle anderen $n - 1 + 1 = n$ Elemente (also alle der Originalliste) enthält.

Fortsetzung: Die Schleifeninvariante bleibt erhalten.
Im Schleifenkörper kann auf $l.head$ zugegriffen werden, da l nicht leer ist. In den folgenden Schritten wird offensichtlich das kleinste Element in der aktuellen Liste l bestimmt, an den Anfang von r angefügt und aus l gelöscht.
Gilt also die Schleifeninvariante für den i-ten Schleifendurchlauf, dann enthält r nach einer weiteren Ausführung des Schleifenkörpers die kleinsten i Elemente. Die

Reihenfolge ist absteigend, denn das neu hinzugefügte Element ist mindestens so groß wie alle schon vorher in r eingefügten Elemente. Das kleinste Element wird folglich aus l entfernt und l enthält wieder die Elemente aus der Originalliste, die nicht in r enthalten sind. Damit gilt die Invariante auch zu Beginn des $(i+1)$-ten Schleifendurchlaufs.

Terminierung: Die Schleife wird offensichtlich n-mal durchlaufen, da in jedem Durchlauf ein Element aus l entfernt wird, l zu Beginn n Elemente enthält und die Schleife abbricht, wenn l leer ist. Wenn also zu Beginn der Schleife zum $(n+1)$-ten mal überprüft wird, ob l leer ist, folgt, dass r die kleinsten $i = n$ Elemente (also alle) von l in absteigender Reihenfolge enthält. r wird zurückgegeben, damit ist SelectionSort korrekt.

(b) Wir gehen davon aus, dass die Eingabeliste n Elemente enthält. Die *while*-Schleife wird so lange ausgeführt, bis l leer ist. In jedem Schleifendurchlauf wird in Zeile 8 ein Element von l entfernt. Daher wird die *while*-Schleife genau n-mal ausgeführt und bei Zeile 3 befinden sich im i-ten Schleifendurchlauf $(i = 1, \ldots, n)$ noch $n+1-i$ Elemente in l.

Die folgende *for*-Schleife wird daher genau $(n+1-i)$-mal durchlaufen, einmal für jedes in diesem Schleifendurchlauf vorhandene Element in l. Das folgende Einfügen in die Liste benötigt bei Verwendung einer linearen Liste $\mathcal{O}(1)$ Zeit und das Entfernen von min von l $\mathcal{O}(n+1-i) = \mathcal{O}(n)$ Zeit. Alle anderen Operationen laufen in konstanter Zeit ab. Damit ergibt sich für die Laufzeit von SelectionSort:

$$\underbrace{\mathcal{O}(n)}_{\textit{while}\text{-Schleife}} \cdot \left(\underbrace{\mathcal{O}(n)}_{\textit{for}\text{-Schleife}} + \underbrace{\mathcal{O}(n)}_{\text{Remove (Zeile 8)}} + \mathcal{O}(1) \right) + \mathcal{O}(1) = \mathcal{O}(n^2)$$

Diese Laufzeit ist für vergleichsbasiertes Sortieren nicht optimal, da z.B. MergeSort in $\mathcal{O}(n \log n)$ sortieren kann.

Der best-case von SelectionSort ist $\Omega(n^2)$: O.B.d.A. nehmen wir an, dass n gerade ist. Dann wird unabhängig vom Listeninhalt die innere *for*-Schleife für die ersten $\frac{n}{2}$ Durchläufe der *while*-Schleife mindestens $(n+1-\frac{n}{2})$-mal durchlaufen und es ergibt sich eine Mindestlaufzeit von:

$$\frac{n}{2}\left(n+1-\frac{n}{2}\right) = \frac{n^2}{4} + \frac{n}{2} = \Omega(n^2).$$

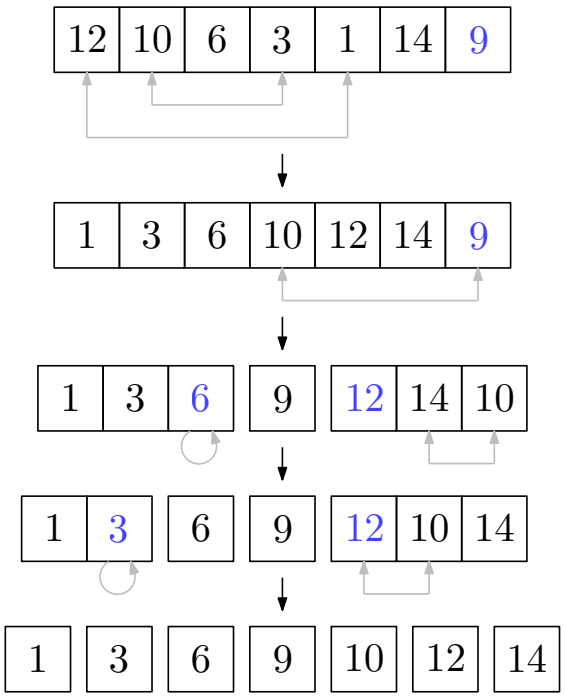
Übung 2.

(a) Sortieren Sie das Array $(12, 10, 6, 3, 1, 14, 9)$ mithilfe des QuickSort-Algorithmus aus der Vorlesung. Wählen Sie hierzu das Pivotelement wie im QuickSort-Algorithmus aus der Vorlesung vorgegeben. Stellen Sie den Inhalt des Arrays nach jedem Aufruf von Partition dar.

(b) Ist der QuickSort-Algorithmus aus der Vorlesung stabil? Begründen Sie Ihre Antwort.

Lösung 2.

(a) Das Pivotelement wird initial am Ende des Arrays ausgewählt, und sonst immer als Nachbarelement des Pivotelements aus dem vorherigen Schritt. Die folgende Grafik illustriert die Ausführung von QuickSort. Pivotelemente werden blau dargestellt. Der Prozess ist parallel dargestellt, obwohl die einzelnen Partition-Aufrufe eigentlich sequenziell abgearbeitet werden. Partition-Aufrufe auf Teilarrays der Länge 1 wurden ignoriert.



(b) Dass QuickSort nicht stabil sein kann, kann man sich an einem Beispiel verdeutlichen: Wir nehmen das Array $(4, 1, 1', 3)$, wobei 1 und $1'$ gleich im Sinne der Ordnungsrelation sind, aber die Reihenfolge beibehalten sollen. Als Pivotelement wird 3 gewählt. Anschließend sucht die die Partition-Funktion von links das erste Element < 3 und rechts das erste Element > 3 und vertauscht diese. Dadurch werden 4 und $1'$ getauscht und wir erhalten das Array $(1', 1, 4, 3)$. Partition vertauscht danach nur noch 3 mit 4; es entsteht das Teilarray $(1', 1)$ und 4. Im ersten Teilarray werden $1'$ und 1 nicht mehr vertauscht und befinden sich danach in falscher Reihenfolge (Swap$(A[i], A[r])$ vertauscht nur das Pivotelement mit sich selbst).

Übung 3.
Geben Sie eine in Zeit $\Theta(n)$ laufende nichtrekursive Prozedur an, die eine einfach verkettete Liste aus n Elementen umkehrt. Stellen Sie sicher, dass Ihre Prozedur höchstens $\mathcal{O}(1)$ zusätzlichen Platz benötigt (neben dem Platz, die für die Eingabeliste gebraucht wird). Begründen Sie, dass Ihre Prozedur korrekt ist und die angegebenen Eigenschaften hat.

Lösung 3.
ReverseList(list l)

```
 1  h := l.head
 2  if h = null
 3        return h
 4  n := h.next
 5  h.next := null
 6  while n ≠ null
 7        x := n.next
 8        n.next := h
 9        h := n
10        n := x
11  return h
```

Wir definieren die Zählvariable des Schleifendurchlaufs i und nutzen als Schleifeninvariante: $\forall i \in \{1, ..., l.length\}$: h zeigt auf aktuellen Listenkopf der umgekehrten Liste für die ersten i Elemente. Die Elemente $[1..i]$ sind in der Reihenfolge umgekehrt.
n ist jeweils der Nachfolger von h in der Originalliste. x wird als Zwischenspeicher verwendet, damit wir den Nachfolger von n in der Originalliste nicht vergessen, bevor wir den *next*-Zeiger von n zu h ändern.

Ist der Listenkopf *null*, so gibt die Prozedur offensichtlich *null* aus.

Initialisierung: Die Schleifeninvariante gilt vor dem Schleifenbeginn: $h.next$ ist gerade *null*, daher zeigt h auf den aktuellen Listenkopf der umgekehrten Liste für das erste Element (h selbst).

Fortsetzung: Im Schleifenkörper wird $n.next := h$ und $h := n$ gesetzt. Dadurch ist die umgekehrte Liste, startend bei h, um 1 länger geworden. Die Zeilen $x := n.next$ und zum Ende $n := x$ stellen sicher, dass n wieder auf den Nachfolger von h in der Originalliste zeigt. Damit sind jetzt die Elemente $[1, ..., i + 1]$ umgekehrt.

Terminierung: Nach Ausführung der Schleife war $n = null$ und n ist der Nachfolger von h in der Originalliste. Daraus folgt, dass h das letzte Element der Originalliste sein muss, welches jetzt der Listenkopf ist. Es wird h zurückgegeben, da dies jetzt der Listenkopf der komplett umgekehrten Liste ist.

Die Prozedur ist offensichtlich nichtrekursiv und läuft in $\Theta(n)$ Zeit, da n in jedem Schritt der *while*-Schleife um eine Stelle vorrückt. Demnach muss n nach $\Theta(n)$ Schritten das Ende der Liste erreichen und *null* werden. Es wird zusätzlich nur Platz für 3 Zeiger (n, h und x) verwendet. Damit wird neben dem Platz für die Eingabeliste $\mathcal{O}(1)$ zusätzlicher Platz benötigt.

Übung 4.

(a) Beweisen Sie: Jeder nichtleere Binärbaum mit k inneren Knoten, in dem jeder innere Knoten zwei Kinder hat, hat $k + 1$ Blätter.

(b) Aus der Vorlesung ist Ihnen bekannt, dass zum Suchen in einem vollständigen Binärbaum $\mathcal{O}(\log_2 n)$ Zeit benötigt wird. Kann die Laufzeit asymptotisch verbessert werden, wenn jeder innere Knoten bis zu 3 Kinder haben darf? Wie sieht es aus, wenn jeder Knoten bis zu $\log_2 n$ Kinder haben darf?

Lösung 4.

(a) Wir zeigen die Aussage per Induktion über k.
Induktionsanfang: $k = 0$. Der Binärbaum ist nicht leer, muss also mindestens einen Knoten enthalten. Mehr Knoten sind nicht möglich, da wir sonst nicht 0 innere Knoten hätten. Der Baum besteht also aus einem Knoten, welcher ein Blatt ist. Damit haben wir $1 = k + 1$ Blätter.
Induktionsannahme: Für ein $k \geq 0$ hat jeder nichtleere Binärbaum mit k inneren Knoten, in dem jeder innere Knoten zwei Kinder hat, $k + 1$ Blätter.
Induktionsschritt: Wir zeigen die Aussage für $k + 1$. Sei also B ein nichtleerer Binärbaum mit $k + 1$ inneren Knoten. Wir wählen einen inneren Knoten v in B mit maximaler Entfernung zur Wurzel des Baums. Seine zwei Kindknoten sind Blätter, sonst wäre die Entfernung von v zur Wurzel nicht maximal. Wir entfernen beide Kindknoten. Dies vermindert die Anzahl der inneren Knoten in B um 1 (da v ein Blatt wird), und die Anzahl der Blätter vermindert sich effektiv um 1. Durch die Entfernung der Knoten bleibt der Restgraph B' offensichtlich ein Baum, in dem jeder innere Knoten zwei Kinder hat. Daher gilt die Induktionsannahme für B'. B' hat k innere Knoten und hat daher $k + 1$ Blätter. Zusammenfassend hat B $k + 1$ innere Knoten und $k + 2$ Blätter.
Induktionsschluss: Damit gilt die Aussage für alle k.

(b) Betrachten wir zunächst den Fall mit 3 Kindern pro innerem Knoten. Da der Baum vollständig ist, ist die Tiefe des Baums $\mathcal{O}(\log_3 n) = \mathcal{O}(\log n)$. Damit wird wie beim Binärbaum $\mathcal{O}(\log_2 n) = \mathcal{O}(\log n)$ Zeit zum Suchen benötigt.

Wenn jeder Knoten bis zu $\log_2 n$ Kinder haben darf, ist die Höhe des vollständigen Baums $\mathcal{O}(\log_{\log_2 n} n)$. Mithilfe der Logarithmengesetze bekommen wir:

$$\mathcal{O}\left(\log_{\log_2 n} n\right) = \mathcal{O}\left(\frac{\ln n}{\ln \log_2 n}\right)$$

In jedem der Suchbaumknoten, den wir während einer Suche durchlaufen, müssen wir entscheiden, in welchen der $\mathcal{O}(\log n)$ Teilbäume wir absteigen müssen. Selbst, wenn wir eine Binärsuche durchführen, wird für diesen Schritt $\mathcal{O}(\log_2 \log_2 n) = \mathcal{O}(\ln \log_2 n)$ Zeit benötigt.

Insgesamt benötigt man also für die Suche $\mathcal{O}(\frac{\ln n}{\ln \log_2 n}) \cdot \mathcal{O}(\ln \log_2 n) = \mathcal{O}(\ln n)$.

In beiden Fällen erhalten wir also keine Laufzeitverbesserung.