Datenstrukturen und Algorithmen: Blatt 7

Bernhard Dietrich (6256800)
Lars Fernhomberg (6256030)
Sebastian Kniesburges (6257120)
Marcus Köthenbürger (6258550)

Übungsgruppe 11
Mittwoch, 11:00-13:00 Uhr in D1.303
Matthias Ernst

Aufgabe	1	2	3	4	Σ	Korrektor
Punkte						
von	6	6	4	4	20	

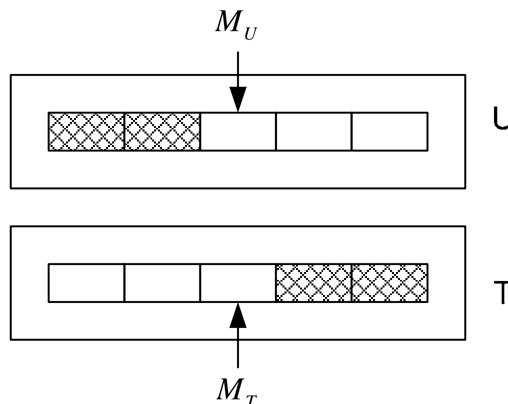
Aufgabe 1

Gegeben sind zwei Arrays U,T mit jeweils n Zahlen in sortierter Reihenfolge. Zeigen Sie, dass der Median der $2n$ Zahlen in den Arrays U,T in Zeit $O(\log n)$ gefunden werden kann.

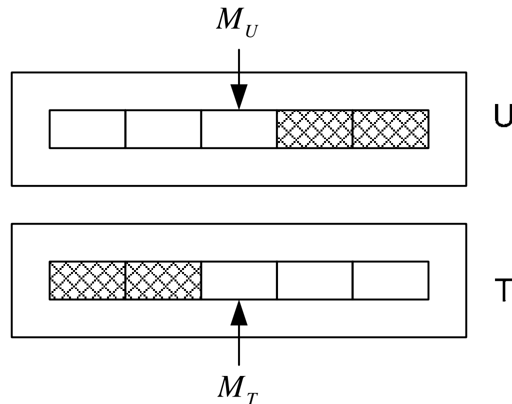
Um den Median der $2n$ Zahlen zu bestimmen, müssen zunächst die beiden Mediane von U und T bestimmt werden, welches dank der (aufsteigenden) Sortierung in konstanter Zeit möglich ist.

Gilt $M_U = M_T$, so ist der Median M_U, da die Gleichheit der beiden Mediane bedeutet, das die Elemente von 0 bis zur Position des Median in beiden Arrays kleiner als ein gemeinsamer Wert und vom Median an größer als ein gemeinsamer Wert sind.

Gilt $M_U < M_T$, so ist bekannt, dass der Median nicht kleiner als M_U und nicht größer als M_T sein kann, so dass sich der Median der $2n$ Elemente zwischen M_U und M_T befinden muss und die anderen Bereiche somit nicht mehr durchsucht werden müssen:



Gilt $M_U > M_T$, so ist bekannt, dass der Median nicht größer als M_U und nicht kleiner als M_T sein, so dass sich der Median der anderen Elemente in den äußeren Teilarrays befinden muss und die anderen Elemente ignoriert werden können:



Die Laufzeit $O(\log n)$ ergibt sich nun daraus, dass der zu durchsuchende Teil des Arrays mit jedem Suchschritt halbiert wird, bis am Ende nur noch ein Element übrig bleibt, welches der Median der beiden Arrays ist.

Aufgabe 2

Wir wollen n Zahlen sortieren. Dabei nehmen wir an, dass die Eingabezahlen paarweise verschieden sind. Entwerfen Sie eine Variante der Paritionsfunktion für Quicksort, so dass Quicksort mit dieser Partitionsfunktion worst-case Laufzeit $O(n \log n)$ besitzt.

Die gewünschte Laufzeit $O(n \log n)$ kann erzielt werden, wenn der Partitionsalgorithmus die beiden vorhandenen Teilarrays so aufspaltet, dass diese auf jeden Fall gut aufgeteilt sind. In der Vorlesung wurde hierfür bestimmt, dass die Laufzeit $O(n \log n)$ bereits erreicht wird, wenn das eine Teilarray 10% und das andere 90% der gesamten Elemente besitzt.
Unsere Idee ist nun, dass man als „Aufteilungselement" statt dem letzten Element des jeweiligen Teilarrays, wie dies bei dem Standard-Partitionsalgorithmus der Fall ist, den Median des Teilarrays ausnutzt. Der Median des Teilarrays liefert eine Verteilung von 50% zu 50% aller Elemente, da alle Elemente vor dem Median kleiner als dieses und alle danach größer als dieses sind. Da die Elemente des Arrays paarweise verschieden sind, gibt es genau ein Element, welches als Median fungiert und somit eine „saubere" Trennung ermöglicht. Ebenfalls kann es nicht zu einem denkbaren worst-case – alle Elemente sind gleich – kommen.
Der Algorithmus lautet somit:

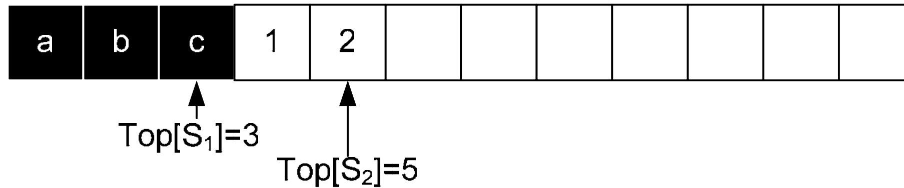
```
MEDIAN-PARTITION(A,p,r)
1.    i ← MEDIAN(A)
2.    A[r] ↔ A[i]
3.    return Partition(A,p,r)
```

Der Aufruf `MEDIAN(A)` in Zeile 1 entspricht dem auf Präsenzübungsblatt 7 Aufgabe 1 gegebenen Algorithmus, der bei einem n-elementigem Array den Median in Zeit $O(n)$ findet.

Die Laufzeit der Zuweisung in Zeile 2 ist konstant $O(1)$ und die Laufzeit des Partitionsalgorithmus beträgt nun, da das gegebene Array genau halbiert wird, somit nur noch maximal $O(n \log n)$ (vgl. Foliensatz „Quicksort - Analyse, Varianten").

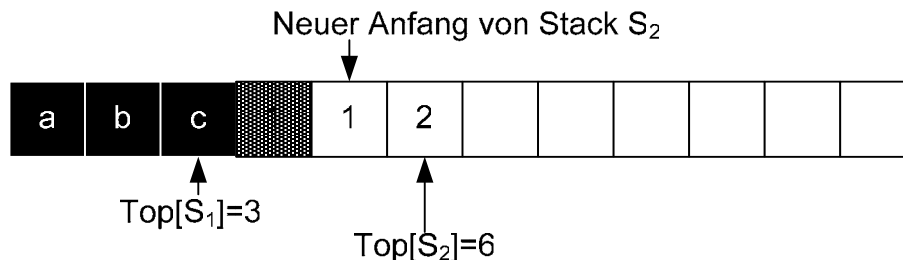
Es ergibt sich somit abschließend $O(n) + O(1) + O(n \log n) = O(n \log n)$.

Aufgabe 3
Zeigen Sie, wie zwei Stacks in einem Array der Größe n realisiert werden können, wenn garantiert ist, dass beide Stacks zusammen nie mehr als n Elemente enthalten.
Das Array hat n Elemente und implementiert die beiden Stacks S_1 und S_2 (zur besseren Verdeutlichung ist der Stack S_1 schwarz markiert und enthält Buchstaben, während der Stack S_2 weiß markiert ist und Zahlen enthält):

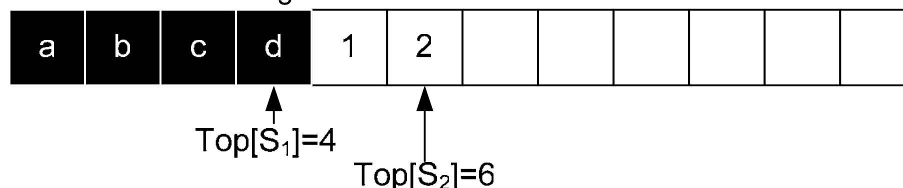


`Top[`S_1`]` ist hierbei das Ende (also das zuletzt hinzugefügte Elemente) des ersten Stacks und `Top[`S_1`]+1` ist der Anfang (und somit das erste hinzugefügte Element) des zweiten Stacks.
Das Einfügen und Löschen von Elementen in S_2 funktioniert wie beim „normalen" Stack, während diese Operationen beim Stack S_1 etwas umständlicher sind.
Beim Einfügen in diesen Stack muss zuerst der Stack S_2 nach hinten verrückt werden, so dass eine „leere" Position entsteht, in der dann das neue Element aus S_2 gespeichert wird:

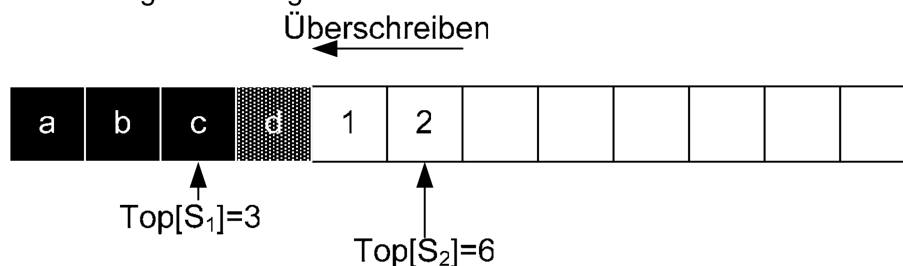
„Platz machen":

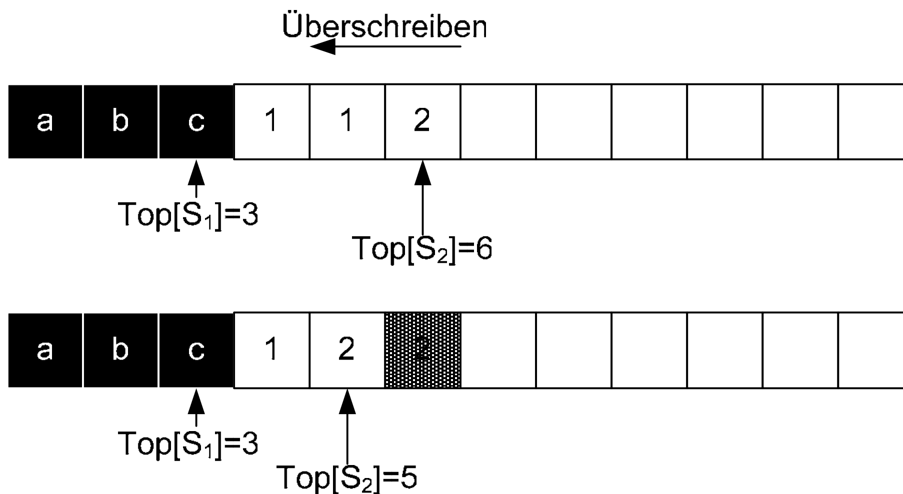


Neues Element einfügen:



Soll nun das oberste Element aus dem ersten Stack herausgenommen werden, so muss das Ende dieses Stacks zuerst um eins zurückgesetzt werden und anschließend müssen die Elemente des zweiten Stacks wieder nach vorne verschoben werden. Dies ist somit die Umkehrung des Einfügens:





Somit ist der ENQUEUE-Algorithmus für den ersten Stack:

```
PUSHS1(S,x)
1.    for i ← top[S2] + 1 downto top[S1] + 2
2.          do S[i] ← S[i - 1]
3.    top[S2] ← top[S2] + 1
4.    top[S1] ← top[S1] + 1
5.    S[top[S1]] ← x
```

Somit ist der DEQUEUE-Algorithmus für den ersten Stack:

```
DEQUEUES1(S)
1.    if STACK-EMPTYS1(S)
2.          then error „Stack underflow!"
3.          else x ← S[top[S1]]
4.                for i ← top[S1] to top[S2] - 1
5.                      do S[i] ← S[i + 1]
6.                top[S2] ← top[S2] - 1
7.                top[S1] ← top[S1] - 1
8.                return x
```

Der Algorithmus um feststellen zu können, ob der zweite Stack leer ist, hat sich ebenfalls verändert, wobei der ursprüngliche Algorithmus aus der Vorlesung für den ersten Stack weiterhin bestand hat:

```
STACK-EMPTYS2(S)
1.    if top[S2] = top[S1]
2.          then return TRUE
3.          else return FALSE
```

Aufgabe 4

Unter dem *Unterlauf* einer Queue versteht man den Versuch, aus einer leere Queue ein Element zu entfernen. Aus dem *Überlauf* einer Queue, realisiert in einem n-elementigen Array, versteht man den Versuch ein weiteres Element in eine Queue einzufügen, wenn die Queue bereits $n-1$ Elemente enthält. Schreiben Sie die Algorithmen ENQUEUE und DEQUEUE so um, dass er auch Unter- bzw. Überlauf einer Queue erkennt und eine Fehlermeldung ausgibt.

Eine leere Queue ist dadurch gekennzeichnet, dass der Zeiger für das Ende (`tail[Q]`) auf die gleiche Position wie der Zeiger für den Anfang der Queue (`head[Q]`) zeigt.
Eine voll besetzte Queue ist dadurch gekennzeichnet, dass der Zeiger für das Ende (`tail[Q]`) auf dem Element vor dem Anfang der Queue (`head[Q]`) liegt.

```
ENQUEUE(Q,x)
1.    if (tail[Q] = head[Q] - 1) OR (tail[Q] = length[Q] AND
            head[Q]=1)
2.                then error „Queue overflow!“
3.                else Q[tail[Q]] ← x
4.                      if tail[Q] = length[Q]
5.                            then tail[Q] ← 1
6.                            else tail[Q] ← tail[Q] + 1

DEQUEUE(Q)
1.    if tail[Q] = head[Q]
2.          then error „Queue underflow!“
3.          else  x ← Q[head[Q]]
4.                if head[Q] = length[Q]
5.                      then head[Q] ← 1
6.                      else head[Q] ← head[Q] +1
7.                return x
```

Hinweis: In der Originalaufgabenstellung war die Formulierung der Aufgabe etwas verwirrend („schreiben Sie den Algorithm**en** ENQUEUE so um, dass er auch Unter- bzw. Überlauf einer Queue erkennt…“), da bei dem Algorithmus ENQUEUE kein Stack-Underflow auftreten kann, da bei diesem Algorithmus lediglich Daten eingefügt werden und keine entfernt werden, so dass es auch nie zu der Situation kommen kann, dass ein nicht vorhandenes Element gelesen werden soll. Bei der Bearbeitung dieser Aufgabe sind wir davon ausgegangen, dass wir die Algorithmen ENQUEUE und DEQUEUE bearbeiten sollen.