Institut für Informatik I
Prof. Dr. Heiko Röglin
Tobias Brunsch

universitätbonn

Algorithmen und Berechnungskomplexität I
Wintersemester 2010/11
Abgabe: 24.11.2010 in der Vorlesung

Übungsblatt 6

Aufgabe 6.1

2+2+2 Punkte

(a) Zeigen Sie, dass *Quicksort* mit der `Partition`-Funktion aus der Vorlesung nicht stabil ist.

(b) Zeigen Sie, dass *Quicksort* mit der `Partition`-Funktion aus Aufgabe 5.2 nicht stabil ist.

(c) Implementieren Sie eine `Partition`-Funktion, mit der *Quicksort* stabil ist. Die `Partition`-Funktion selbst soll in-place sein und Zeit $O(n)$ benötigen, wobei n die Länge des Feldes ist.

Aufgabe 6.2

2+1+3 Punkte

Rekursionen werden über einen Stack realisiert, auf dem die Parameter und die lokalen Variablen der einzelnen Funktionsaufrufe gespeichert werden. Wenn ein Aufruf abgearbeitet wurde, werden die entsprechenden Daten vom Stack entfernt und es wird zu der Codezeile zurückgesprungen, von der aus der Aufruf erfolgte. Wenn man den Rekursionsstack selbst verwaltet, kann man auf rekursive Aufrufe verzichten. Bei manchen Problemen sind dadurch Optimierungen der Stackverwaltung möglich.

(a) Betrachten Sie eine rekursive Funktion `Func`, bei der alle rekursiven Aufrufe am Ende erfolgen.

```
Func(Parameterliste)
{
  // (1) Anweisungen, die keinen Aufruf von Func verursachen

  ...

  // (2) Rekursive Aufrufe von Func
  Func(Parameterliste_1)
            ⋮
  Func(Parameterliste_k)
}
```

Beschreiben Sie, wie man die Stackverwaltung für den Aufruf von `Func` so optimieren kann, dass weniger Speicher benötigt wird.

(b) *Quicksort* hat die Gestalt von `Func` aus Aufgabenteil (a). Bei der herkömmlichen Implementierung der Stackverwaltung wird für den Stack im worst-case $\Theta(n)$ Speicher benötigt, wobei n die Größe des zu sortierenden Feldes ist. Wir wollen *Quicksort* in-place implementieren, d.h. wir dürfen nur $O(\log n)$ zusätzlichen Speicher verwenden. Dies gilt insbesondere für den Stack. Wie kann das erreicht werden?

Hinweis: Die Idee aus Aufgabenteil (a) allein reicht noch nicht.

(c) Beweisen Sie, dass Ihre Implementierung von *Quicksort* in-place ist.

Aufgabe 6.3

1+1+2+2 Punkte

Wir erweitern das Konzept des *binären Heaps* von binären Bäumen auf Bäume mit Verzweigungsgrad d, d.h. Bäume, deren Knoten bis zu d Kinder besitzen. Ein d-Heap ist ein Baum mit Verzweigungsgrad d, der die *max-Heap*-Eigenschaft erfüllt, d.h. der Schlüssel jedes Knotens ist nicht kleiner als die Schlüssel aller seiner Kinder (vgl. Abbildung 1).

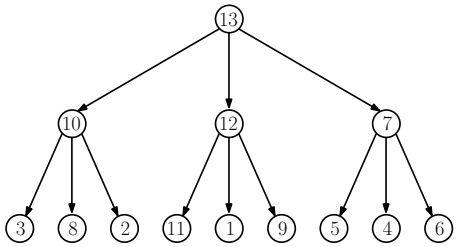

Abbildung 1: Ein 3-Heap mit 13 Knoten

(a) Stellen Sie den im folgenden Feld repräsentierten 3-Heap als Baum dar.

40	32	24	36	28	30	5	12	21

(b) Wir betrachten die Felddarstellung eines d-Heaps. Geben Sie die Formel an, mit der die Positionen der Kinder des Elementes an Position i bestimmt werden können. Die Nummerierung der Feldeinträge soll bei 0 beginnen.

(c) Geben Sie eine modifizierte Version des in der Vorlesung vorgestellten Algorithmus `Reheap` an, der Elemente in einem d-Heap versickern lässt. Der Grad d wird hierbei als Parameter übergeben.

(d) Sortieren Sie das in Aufgabenteil (a) gegebene Feld entsprechend der *Heapsort*-Methode für 3-Heaps. Zeichnen Sie das Feld nach jeder Vertauschung.

Aufgabe 6.4

1+1+1+1+1+1 Punkte

Bei der Kommunikation über Netzwerke kann nicht immer sichergestellt werden, dass die einzelnen Pakete beim Empfänger in der gleichen Reihenfolge ankommen, wie sie versendet wurden. Um dadurch entstehende Probleme zu lösen, wird jedes Paket mit einer eindeutigen, aufsteigenden ID versehen. Eine Veränderung der Reihenfolge ist jedoch nicht der Regelfall, sondern eher die Ausnahme. Darüber hinaus können wir davon ausgehen, dass kein Paket von vielen Paketen überholt wurde. Um die ursprüngliche Reihenfolge der Pakete wiederherzustellen, werden diese nach dem Empfangen aller Pakete auf der Empfängerseite sortiert.

Geben Sie für jeden der folgenden Sortieralgorithmen an, ob er für dieses Problem geeignet ist. Begründen Sie jeweils Ihre Antwort.

(a) *Quicksort*

(b) *Bubblesort* mit frühzeitigem Abbruch (vgl. Aufgabe 5.1)

(c) *Selectionsort* (vgl. Aufgabe 1.4)

(d) *Heapsort*

(e) *Insertionsort*

(f) *Insertionsort* mit binärer Suche zur Bestimmung der Einfügeposition

Aufgabe 6.5

Bitte geben Sie auf einem Extrazettel (anonym) Feedback zu den bisherigen Vorlesungen und Übungen. Was hat Ihnen gefallen und was sollte aus Ihrer Sicht verbessert werden? Erwähnen Sie bei Bemerkungen zur Übung ggf. Ihre Übungsgruppe.