Algorithmieren und Programmieren

Sommersemester 2023

Lehrstuhl Programmiersprachen und Compilerbau
Prof. Dr. rer. nat. habil. Petra Hofstedt
Ilja Becker
Andreas Eberle, Jan Robert Meyer, Oliver Pflug

Brandenburgische
Technische Universität
Cottbus - Senftenberg

Übungsblatt 11

Abgabedatum: 16.07.2023

Hinweise

- Beachten Sie, dass die Tutoren unkommentierte Programme nicht als Lösung akzeptieren, selbst wenn die Programme richtig funktionieren. Zu einer richtigen Lösung gehören immer sinnvolle Kommentare, deren Umfang der Komplexität des Programms angemessen ist.
- **Halten Sie sich an die in den Übungsblättern vorgegebenen Namen von Funktionen und Dateien, Funktionstypen (Typsignaturen), Reihenfolge der Parameter und verwenden Sie — sofern vorhanden — die Vorgaben!**
- Auf den Übungsblättern finden Sie einige Haskell-Quelltextfragmente. Diese sind, der besseren Lesbarkeit wegen, unter Nutzung einiger mathematischer Sonderzeichen wiedergegeben.
- Für diese Veranstaltung wird die Verwendung des ***Glasgow Haskell Compiler (GHC)*** (`https://www.haskell.org/`) empfohlen. Weitere Informationen finden Sie im Software-Reiter auf Moodle.
- Als **Tutoriumsaufgabe** markierte (Teil-)aufgaben werden in den Übungen ausführlicher besprochen. Die schriftlichen bzw. elektronischen Lösungen müssen jedoch trotzdem mit abgegeben werden. Bitte schauen Sie Sich diese Aufgaben im Vorfeld der Übung an und bereiten Sie Sich darauf vor.
- Die Abgabe Ihrer Lösungen erfolgt vor Ablauf der Abgabefrist digital über die Moodle-Plattform an Ihren Tutor. Erstellen Sie dazu ein PDF-Dokument, das die Lösungen Ihrer schriftlichen Aufgaben enthält. Laden Sie dieses PDF-Dokument und die erzeugten Hakell-Dateien, mit den in den Aufgaben vorgegebenen Namen, bei Moodle hoch.

Informationsquellen

- Sie finden unter `http://haskell.org/` sehr viele Informationen über die Programmiersprache Haskell. Von besonderem Interesse sind dabei sicherlich die Übersicht über zahlreiche online verfügbare Haskell-Tutorials (`https://wiki.haskell.org/Tutorials`) sowie die Suchmaschine Hoogle (`http://hoogle.haskell.org/`) für die Haskell-API, die Ihnen mit zunehmender Haskell-Erfahrung wertvolle Dienste leisten wird.

Sie können maximal **(7 Punkte)** mit diesem Übungsblatt erreichen.

Aufgabe 1 (Heapsort mit Max-Heap) 3 Punkte

(Tutoriumsaufgabe) Die folgenden Werte liegen einem 11-elementigen Array vor:

Index:	0	1	2	3	4	5	6	7	8	9	10
Wert:	„PM“	„LOS“	„BRB“	„CB“	„SPN“	„FF“	„EE“	„TF“	„LDS“	„P“	„OSL“

Dieses Array soll nach lexikographischer Ordnung[1] aufsteigend mittels Heapsort sortiert werden. Wir wollen nachfolgend die dazu notwendigen Arbeitsschritte veranschaulichen.

1. **(0,5 Pkt.)** Überführen Sie das Array in eine Darstellung als binärer Baum, so dass das Element mit Index 0 zum Wurzelknoten wird. Darüber hinaus hat jeder Knoten mit Elementindex i die Nachfolger mit den Elementindizes $2i+1$ (linker Nachfolger) und $2i+2$ (rechter Nachfolger), soweit vorhanden. Man erreicht dies durch fortlaufend ebenenweises Eintragen der Arrayelemente in den zugehörigen binären Baum.
2. **(1 Pkt.)** Aus dem anfänglichen binären Baum wird als erster Schritt von Heapsort ein Max-Heap aufgebaut. Dazu durchlaufen wir nacheinander von unten nach oben alle Teilbäume und wandeln sie durch Versickern der *kleinsten* Knoten in Max-Heaps um. Im Ergebnis steht das jeweils größte Teilbaumelement an der entsprechenden Wurzelposition. Identifizieren Sie rechts unten beginnend nacheinander die einzelnen Teilbäume, und veranschaulichen Sie die Veränderungen der Knoten. Prüfen Sie anschließend, dass der gesamte Binärbaum einen Max-Heap darstellt.
3. **(1,5 Pkt.)** Im zweiten Schritt von Heapsort wird abwechselnd der Wurzelknoten des Max-Heaps mit dem rechtesten Blatt der untersten Ebene vertauscht und durch Versickern des neuen Wurzelknotens wieder ein Max-Heap hergestellt, bis der gesamte Baum abgearbeitet ist. In jedem Durchlauf wird dabei das jeweils größte Baumelement herausgefunden und in den sortierten Teil des Arrays übernommen. Der noch zu bearbeitende Baum vekleinert sich dabei um einen Knoten. Illustrieren Sie die einzelnen Arbeitsschritte.

Aufgabe 2 (Min-Heap ADT) 4 Punkte

In dieser Aufgaben soll ein Modul `MinHeap` implementiert werden, dass einen binären Heap implementiert. Hierbei soll die interne Umsetzung mit einer Liste umgesetzt werden. Die Liste steht hierbei stellvertretend für ein Array, auch wenn es keinen Zugriff in $O(1)$ auf beliebige Elemente zulässt. Das Modul soll folgende Funktionen/Datentypen implementieren:

- `MinHeap a`
  Datentyp, der auf Basis einer Liste einen Min-Heap repräsentiert
- `emptyMinHeap :: Ord a ⇒MinHeap a`
  Erzeugt einen leeren `MinHeap`
- `heapEmpty :: Ord a ⇒MinHeap a →Bool`
  Überprüft, ob ein `MinHeap` leer ist

[1] Lexikographische Ordnung bedeutet, dass Zeichenketten (Strings) nach der Stellung eines Buchstabens im Alphabet sortiert werden. Kommt der erste Buchstabe c_1 einer Zeichenkette str_1 im Alphabet vor dem ersten Buchstaben c_2 einer zweiten Zeichenkette str_2, also $c_1 < c_2$, dann ist $str_1 < str_2$. Sind die Buchstaben gleich ($c_1 = c_2$), so wird der nächste Buchstabe in der Zeichenkette betrachtet. Ist ein Wort in Gänze als Anfangsteil in einem anderen enthalten, so gilt ist das kürzere Wort kleiner als das Größere. (Siehe auch `https://www.wikiwand.com/de/Lexikographische_Ordnung`.)

- `checkMinHeapProperty :: Ord a ⇒MinHeap a →Bool`
  Überprüft, ob in einem Heap die Min-Heap-Eigenschaft erfüllt ist, also die Werte aller Elternknoten stets kleiner als die ihrer Kindknoten sind
- `minElem :: Ord a ⇒MinHeap a →a`
  Gibt das kleinste Element im Min-Heap zurück
- `insMinHeap :: Ord a ⇒a →MinHeap a →MinHeap a`
  Fügt ein Element in den Min-Heap ein und stellt bei Bedarf die Min-Heap-Eigenschaft wieder her
- `delMinHeap :: Ord a ⇒MinHeap a →MinHeap a`
  Entfernt das minimale Element (also die Wurzel) aus dem Min-Heap und stellt die Min-Heap-Eigenschaft wieder her

Bitte testen Sie wie immer ihre Funktionen.

a) **(1 Pkt.)** Legen Sie zunächst ihr Modul `MinHeap` an und implementieren Sie den listenbasierten Datentyp sowie die Funktionen `emptyMinHeap` und `heapEmpty`. Machen Sie sich vertraut mit der Funktion `(!!)`[2]. Es ist außerdem empfehlenswert (wenn auch nicht zwingend notwendig), Hilfsfunktionen anzulegen, die z.B. für einen Heap h und einen Index i wenn vorhanden den Elternknoten, den linken oder rechten Kindknoten liefern oder deren Index in Relation zum gegebenen Knoten i berechnen. Es ist außerdem hilfreich, eine Funktion `swap :: Int →Int →[a] →[a]` (alternativ: `Integer` statt `Int`) zu implementieren, die zwei Indizes sowie eine Liste übergeben bekommt und die Elemente an den Indizes in der Liste vertauscht.

b) **(1 Pkt.)** Implementieren Sie die Funktion `checkMinHeapProperty`, die für jeden Knoten im Heap überprüft, ob beide Kinder (sofern Kinder vorhanden sind) größer sind.

c) **(1 Pkt.)** Implementieren Sie die Funktion `insMinHeap`, die ein Element in den Min-Heap einfügt und bei Bedarf die Min-Heap-Eigenschaft wiederherstellt. Zum Einfügen eines Elements in einen Heap geht man folgendermaßen vor[3]:

   1) Füge das Element in der „untersten Ebene“ soweit wie möglich links ein - d.h. am Ende des Arrays, bzw. in diesem Fall am Ende der Liste.
   2) Vergleiche das neu hinzugekommene Element mit seinem Elternknoten[4]
      - Ist das neue Element größer als der Elternknoten und die Heap-Eigenschaft damit erfüllt, ist der Einfüge-Prozess abgeschlossen.
      - Ist das neue Element kleiner als der Elternknoten, so vertausche die beiden Elemente. Betrachte nun den neuen Elternknoten des neuen Elements auf dieselbe Art. Wiederhole, bis die Heap-Eigenschaft erfüllt ist (oder das neue Element der neue Wurzelknoten ist).

d) **(1 Pkt.)** Implementieren Sie die Funktion `delMinHeap`, die das minimale Element des Heaps entfernt und wieder einen intakten Heap erstellt. Wenn Sie das minimale Element eines Min-Heaps entfernen wollen, so müssen Sie die Wurzel entfernen[5]. Daraufhin müssen Sie den

[2] s. Haskell Dokumentation zu Indexing in Listen (`https://hackage.haskell.org/package/base-4.14.0.0/docs/Data-List.html#g:16`).

[3] s. auch: `https://en.wikipedia.org/wiki/Binary_Heap`

[4] Für einen Max-Heap gälten die Vergleiche entsprechend andersherum.

[5] In der Vorlesung wird das Verfahren etwas anders beschrieben. Dort wird das Wurzelelement mit dem letzten Element im Array vertauscht, und dann einfach nicht weiter betrachtet. Dies ist dadurch begründet, dass die Größe des Arrays fest steht und vielmehr festgehalten wird, welches das letzte zu betrachtende Element im Array ist. Da wir hier mit einer Liste arbeiten, ist dies nicht notwendig und wir können die Wurzel einfach entnehmen/verwerfen.

Min-Heap wieder in Ordnung bringen, indem Sie die beiden Teilbäume mit einer neuen Wurzel verbinden und die Min-Heap-Eigenschaft wieder herstellen. Dies geht folgendermaßen:

1) Ersetze die Wurzel mit dem am weitesten unten rechts liegenden Element, also dem Element am Ende des Arrays bzw. in diesem Fall der Liste.
2) Versickere die neue Wurzel im Baum:
    i) Vergleiche das neue Wurzelelement mit seinen Kindern und vertausche es bei Bedarf mit dem kleineren der beiden Elemente.
    ii) Wiederhole dies solange, bis die Min-Heap-Eigenschaft wiederhergestellt ist.