Übungen zur Vorlesung

Algorithmen und Datenstrukturen

Übungsblatt 4

Cryptoplexity
Cryptography & Complexity Theory
Technische Universität Darmstadt
www.cryptoplexity.de

Arbeitsgruppe Kryptographie und Komplexitätstheorie
Prof. Dr. Marc Fischlin
Dr. Christian Janson
Patrick Harasser
Felix Rohrbach

Sommersemester 2019
Veröffentlicht am: 10.05.2019
Abgabe am: 24.05.2019, 12:00 Uhr

P1 (Gruppendiskussion)

Nehmen Sie sich etwas Zeit, um die folgenden Fachbegriffe in einer Kleingruppe zu besprechen, sodass sie anschließend in der Lage sind, die Begriffe dem Rest der Übungsgruppe zu erklären:

(a) Stacks, Queues und deren Operationen
(b) Verkettete Listen und deren Operationen
(c) Bäume und deren Operationen
(d) Inorder-, Preorder- und Postorder-Traversierungen

P2 (Stacks und Queues)

(a) Im folgenden soll eine Queue Q wie in der Vorlesung auf einem Array dargestellt werden. Die Größe des Arrays ist dabei 4. Stellen Sie das Array nach jeder der folgenden Operationen dar: `enqueue`$(Q, 3)$; `enqueue`$(Q, 4)$; `dequeue`(Q); `enqueue`$(Q, 6)$; `enqueue`$(Q, 7)$; `enqueue`$(Q, 8)$; `dequeue`(Q); `dequeue`(Q).
(b) Realisieren Sie eine Queue mithilfe von zwei Stacks. Implementieren Sie dazu in Pseudocode die Methoden `new`(Q), `isEmpty`(Q), `enqueue`(Q, x) und `dequeue`(Q). Analysieren Sie die asymptotische Laufzeit der vier Methoden.
(c) Realisieren Sie einen Stack mithilfe von zwei Queues. Implementieren Sie dazu in Pseudocode die Methoden `new`(S), `isEmpty`(S), `push`(S, x) und `pop`(S). Analysieren Sie die asymptotische Laufzeit der vier Methoden.

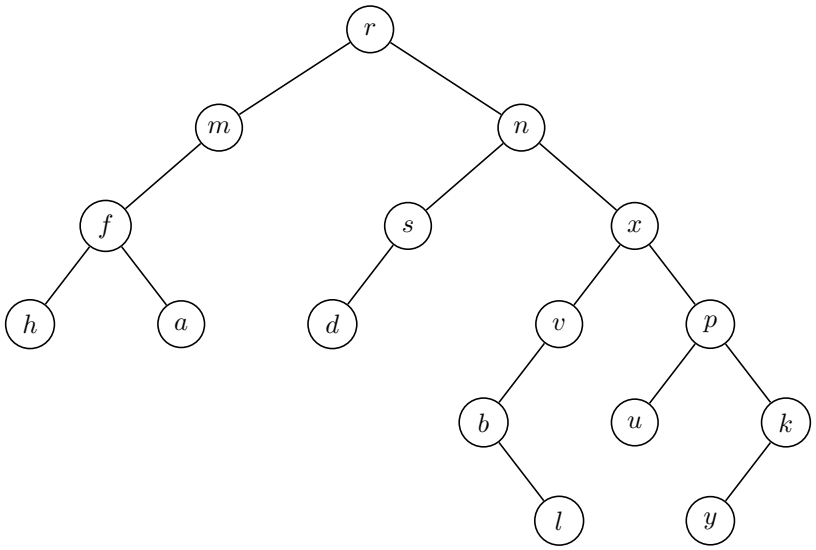
P3 (Linked lists)

Eine *einfach verkettete Liste* ist eine Datenstruktur, die ähnlich definiert wird wie eine doppelt verkettete Liste, wo aber jedes Element nur den Zeiger auf das nachfolgende Element in der Liste besitzt (es fehlt also der Zeiger auf das vorhergehende Element).

(a) Vergleichen Sie einfach und doppelt verkettete Listen, indem Sie Vorteile der jeweiligen Datenstruktur beschreiben.
(b) Formulieren Sie einen nicht rektursiven Algorithmus, der eine einfach verkettete Liste mit n Elementen umkehrt. Die Methode soll keinen zusätzlichen Speicher wie extra Listen benutzen (es sind nur temporäre Hilfsvariablen, die unabhängig von der Größe n der Liste sind, erlaubt). Die Laufzeit des Algorithmus soll in $O(n)$ liegen. Beweisen Sie auch die Korrektheit Ihres Algorithmus.

P4 (Bäume)

Betrachten Sie folgenden Baum:



(a) Welcher Knoten ist die Wurzel des Baumes?
(b) Geben Sie die Vorfahren von v und die Nachkommen von x an.
(c) Geben Sie die Kinder von n und die Elternknoten von f an.
(d) Welche sind die Geschwister von d?
(e) Was ist die Tiefe von b?
(f) Wie viele innere Knoten und Blätter hat der Baum?
(g) Geben Sie den rechten Teilbaum von x an.

P5 (Binäre Suchbäume)

(a) Fügen Sie nacheinander Knoten mit den Schlüsseln $17, 10, 1, 29, 7, 20, 30, 23$ in einen leeren binären Suchbaum ein. Skizzieren Sie den resultierenden Baum.
(b) Traversieren Sie den Baum jeweils nach dem Inorder-, Preorder-, und dem Postorder-Verfahren. Fällt Ihnen bei einem etwas auf?
(c) Entfernen Sie nun nacheinander die Knoten mit den Schlüsseln $1, 23, 29$. Geben Sie den entstehenden Baum, sowie für jeden Löschvorgang den zutreffenden Fall des Löschens an.

H1 (Rot-Schwarz-Bäume) (10+5 Punkte)

In dieser Übung sollen Sie lernen, Bäume selbst zu implementieren. **Bitte lesen Sie sich zunächst die allgemeinen Hinweise für die praktischen Übungen auf Moodle durch!** Laden Sie sich anschließend das vorgegebene Framework herunter, um die Aufgabe implementieren zu können.

(a) Implementieren Sie einen Rot-Schwarz-Baum. Vervollständigen Sie dazu die Funktionen `search`, `maximumSubtree`, `minimumSubtree`, `insert` und `delete` in der Klasse `RedBlackTree`.

(b) Der "Completely Fair Scheduler", Prozess-Scheduler für den Linux-Kernel, der bestimmt, welcher Prozess als nächstes ein Ausführungsfenster auf der CPU bekommt, benutzt intern einen Rot-Schwarz-Baum. Sie sollen eine stark vereinfachte Variante dieses Schedulers programmieren, die wie folgt funktioniert:

- Der Wert jedes Prozesses ist die (summierte) Ausführungszeit, die er bisher auf der CPU bekommen hat. Diese kann über `Process.executionTime()` abgefragt werden.
- Dem Prozess kann ein Ausführungsfenster gegeben werden, indem `Process.execute()` ausgeführt wird. Nach jeder Ausführung wird der Prozess aus dem Baum entfernt. Falls der Prozess noch nicht abgeschlossen ist (überprüfbar per `Process.finished()`), wird er mit der neuen Ausführungszeit im Baum eingefügt. Existiert schon ein Prozess mit dieser Ausführungszeit, wird die Ausführungszeit so lange um eins erhöht, bis es noch keinen Prozess mit dieser Ausführungszeit im Baum gibt.
- Der Scheduler hat die Methoden `run(int windows)` und `add(Process p)`. `run` weißt den Scheduler an, die angegebene Anzahl an Ausführungsfenstern an die Prozesse zu verteilen. Der Scheduler wählt dabei immer den Prozess mit der kleinsten Ausführungszeit. Sollten alle Prozesse fertig sein, verfallen die weiteren Fenster. `add` fügt einen neuen Prozess in den Baum ein. Auch hier wird die Ausführungszeit so lange hochgezählt, bis noch kein Prozess mit dieser Ausführungszeit im Baum vorhanden ist. Weiterhin hat der Scheduler eine Methode `finished`, die angibt, ob alle Prozesse durchgelaufen sind (und der Baum damit leer ist).

Implementieren Sie die Methoden `run`, `add` und `finished` in der Klasse `CompletelyFairScheduler`.