Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik

Prof. Dr. W.-P. de Roever
Dr. M. Steffen

Algorithmen & Datenstrukturen

Wintersemester 2000/2001 | 14. November 2000

Serie 4

Thema : Bäume

Ausgabetermin: 14. November 2000

Abgabe: 23. November 2000

Aufgabe 1: [Bonus: Sortieren]

Zum Nachdenken: Im Zeitalter der Lochkarten gab (und gibt) es folgendes Sortierverfahren: Die Lochkarten sollen gemäß eines *Sortierschlüssel* sortiert werden. Der Sortierschlüssel ist an der *oberen Kante* der Karten in binärer Form in einer horizontalen Reihe von Ausstanzungen codiert und zwar auf folgende Weise: eine binäre 0 sei durch ein Loch kodiert und eine 1 durch eine U-förmige, nach oben offene Aussparung. Legt man einen Stapel zu sortierender Karten ordentlich hintereinander, so kann man an einer festen Position der Binärkodierung mit einer Stricknadel durch alle Löcher bzw. Aussparungen stoßen.

Sortiert wird nun mittels folgenden Algorithmusses

1. Stoße mit der Nadel durch ein Loch.
2. Hebe die Nadel nach oben.
3. Stecke alle hängengebliebenen Karten hinter die im Kasten gebliebenen.
4. Wiederhole die ersten 3 Schritte, bis alle Löcher/Binärstellen behandelt sind.

Ich welcher Reihenfolge müssen die Stellen abgearbeitet werden? Entspricht/Ähnelt dieses Verfahren einem der in der Vorlesung vorgestellten Verfahren? Die Komplexität des Verfahrens ist offensichtlich $O(\log n)$. Das ist *extrem* (!) gut (nur mit Tricks haben wir in der Vorlesung *lineare* Verfahren hinbekommen.) Warum ist das Verfahren nur unter Lochkartensortierern bekannt, aber nicht sehr unter Programmierern?

Aufgabe 2: [Listen]

- Geben Sie die Komplexität folgender Operationen für verschiedene Repräsentierungen von Listen an: sortiert oder unsortiert, einfach-verzeigert oder doppelt.

Operation	unsortiert, einfach	sortiert, einfach	unsortiert, doppelt	sortiert, doppelt
$search(L, k)$				
$insert(L, x)$				
$delete(L, x)$				
$successor(L, x)$				
$predecessor(L, x)$				
$minimum(L)$				
$maximum(L)$				

- Eine beliebte Repräsentierung für (doppelt) verzeigerte Listen verwendet einen sogenannten "*sentinel*" oder Wächter, ein Dummy-element am Beginn der Liste. Zusätzlich verzeigert man das Ende der Liste so, daß der Nachfolger des letzten Elementes wieder die Dummy-Zelle ist (d.h., es handelt sich um eine doppelt-verzeigerte, zirkuläre Liste).

  Implementieren Sie die Operationen *List-insert*, *List-delete* und *List-search* auf dieser Struktur. Argumentieren Sie, warum man dieses Dummy-Element oft einführt.

Aufgabe 3: [Suchbäume mit Lifo-Eigenschaft]

Gefordert seien *binäre Suchbäume* mit folgenden drei Wörterbuchoperationen

1. $insert(key, elem)$,
2. $delete(key)$ und
3. $search(key)$,

wobei das Löschen einen der Knoten mit dem passenden Schlüssel, falls vorhanden, löscht und das Suchen, falls erfolgreich, das zugehörige Datenelement zurückliefert. Die Schlüssel im Baum brauchen dabei nicht eindeutig sein, es kann also mehrere Knoten mit dem selben Schlüssel geben.

Bei mehrfachem Vorkommen von Schlüsseln soll zusätzlich folgende *Disziplin* eingehalten werden: die Suche nach einem Schlüssel fördert immer dasjenige passende Element zu Tage, welches zu diesem Schlüssel *zuletzt* eingefügt wurde und das Löschen betrifft dasselbe Element. Mit anderen Worten, die Elemente zu einem festen Schlüssel unterliegen einer *last-in-first-out*-Disziplin.[1]

Ändern Sie die vorgestellten binären Suchbäume geeignet ab und beschreiben Sie, warum Ihre Baumimplementierung die Lifo-Eigenschaft besitzt.

[1] Der Baum verhält sich also nach außen hin wie eine Hashtabelle mit externer Verkettung. Derartige Strukturen werden beispielsweise dazu verwendet, um *Symboltabellen* zu repräsentieren, d.h., die Assoziation von Identifiern zu Werten und ähnlichem, wobei die Stackeigenschaft für die geschachtelten Gültigkeitsbereiche der Identifier benötigt wird.