Prof. Dr. Gerd Stumme, Folke Eisterlehner, Dominik Benz
Fachgebiet Wissensverarbeitung

06.07.2009

13. Hausübung „Algorithmen und Datenstrukturen“

Sommersemester 2009

Abgabetermin: Montag, 13.07.2009, 10:00 Uhr

Digitale Bäume

Aufgabe 1 (Lexikographisches Sortieren mit Tries)

In der Vorlesung wurden *Tries* als spezielle Suchbäume vorgestellt, die Zeichenketten variabler Länge enthalten. Beschreiben Sie einen Algorithmus, mit dem alle Zeichenketten, die in einem gegebenen Trie enthalten sind, in lexikographisch sortierter Reihenfolge ausgegeben werden.

(3 Punkte)

Aufgabe 2 (Vergleich Tries / Patricia-Bäume)

Folgende Zeichenketten-Mengen (bestehend aus jeweils 5 Zeichenketten) sollen jeweils in einem Trie und eine Patricia-Baum abgespeichert werden:

a) `Fachbildungsprojekt`, `Dialogschwerpunkt`, `Komplizenschaft`, `unproblematisch`, `Zwischenprodukt`

b) `1000000`, `1001000`, `1001010`, `1000011`, `1001100`

c) `Haus`, `Hausarbeit`, `Hausaufgabe`, `Hausaufgabenbetreuung`, `Hausarbeitsthema`

Erstellen Sie für jede der drei Zeichenketten-Mengen jeweils einen Trie und einen Patricia-Baum. (*Hinweis:* Verwenden Sie – wenn nötig – das Zeichen **`$`** um das Ende einer Zeichenkette darzustellen.) Lesen Sie anschließend aus diesen Bäumen folgende Werte ab:

- Gesamtanzahl der Knoten (**`nodeCount`**). Mit “Knoten” sind hierbei *alle* Knoten im Baum gemeint, d.h. alle inneren Knoten und die Blattknoten. So besteht z.B. der Patricia-Baum auf Folie 6-74 aus genau 8 und der Trie von Folie 6-73 aus genau 16 Knoten.
- Mindestanzahl der Vergleiche, die zum Finden einer Zeichenkette nötig sind (**`minComp`**)
- Maximale Anzahl der Vergleiche, die zum Finden einer Zeichenkette nötig sind (**`maxComp`**)
- Durchschnittliche Anzahl der Vergleiche, die nötig sind, wenn nach jeder Zeichenkette im Baum genau einmal gesucht wird (**`avgComp`**)

Fassen Sie Ihre Ergebnisse in einer Tabelle der folgenden Form zusammen:

		`nodeCount`	`minComp`	`maxComp`	`avgComp`
a)	Trie				
	Patricia-Baum				
b)	Trie				
	Patricia-Baum				
c)	Trie				
	Patricia-Baum				

Hinweis: Als Abgabe genügt uns eine ausgefüllte Tabelle in dieser Form; Sie brauchen *keine* graphische Version Ihrer Tries / Patricia-Bäume mit abzugeben.

(9 Punkte)

Hash-Verfahren

Aufgabe 3 (Arbeit mit Hashtabellen)

Gegeben sei die Schlüsselfolge $a = 24, 15, 42, 16, 12, 27, 18$. Fügen Sie diese sukzessive in

a) eine Hashtabelle t_1 der Größe $m_1 := 7$

b) eine Hashtabelle t_2 der Größe $m_2 := 9$

ein. Die Tabellen Seien anfänglich leer, wobei ein leerer Eintrag durch „_“ dargestellt wird.

Verwenden Sie $h_i(x) := x \bmod m_i$ als *Hashfunktion* ($i = 1, 2$) und *lineares Sondieren* zur *Kollisionsbehandlung*: Kommt es beim Einfügen eines Elements k in t_i zu einer Kollision (d.h. $t_i[h_i(k)] \neq \_$), so wird für $j = 1, 2, \ldots$ fortlaufend

$$(h_i(k) + j) \bmod m_i$$

als neuer Hashwert berechnet, bis ein freier Platz gefunden wird.

Geben Sie den Inhalt der Hashtabelle nach der jeweils letzten Einfügeoperation an und notieren Sie in einer Tabelle zu jedem Schlüssel die Anzahl der jeweils benötigten Sondierungsschritte.

Der Inhalt einer Hashtabelle *t* soll dabei wie folgt angegeben werden:

```
0: -
1: a3
2: -
3: -
4: a2
```

(*t* enthält in Postion 1 den Wert a_3, an Postion 4 den Wert a_2 und ist ansonsten leer) (8 Punkte)

Heap-Sort

Aufgabe 4 (Sortierreihenfolge bei Heap-Sort)

In der Vorlesung wurde der Algorithmus *Heap-Sort* vorgestellt, der die in einem Array gegebene Schlüsselfolge in Zeit $O(n \log n)$ absteigend sortiert.

Wie kann man den Algorithmus am einfachsten modifizieren, so dass die Schlüsselfolge aufsteigend sortiert wird? (3 Punkte)

Aufgabe 5 (Implementierung Heap-Sort)

Auf der Vorlesungsseite finden Sie eine Implementierung des Heap-Sort-Algorithmus, wie Sie auf Folie 6-87 vorgestellt ist (**`HeapSort.java`**). Diese basiert auf einer Klasse **`Heap.java`**, die die Daten des Heaps speichert und Methoden zur Erstellung des Heaps zur Verfügung stellt:

- **`heapify()`**: Erstellt einen Heap aus den gespeicherten Daten
- **`siftDown(int start, int end)`**: Lässt das Element, das an Stelle **`start`** gespeichert ist, im Heap "versickern", maximal bis zur Stelle **`end`**
- **`swap(int index1, int index2)`**: Vertauscht die Werte an den Stellen **`index1`** und **`index2`**.

Die Klasse **`Heap.java`**, die sie ebenfalls auf der Vorlesungsseite finden, stellt ein Gerüst dar, in dem diese Methoden zwar angelegt, aber nicht implementiert sind. Programmieren Sie die Methoden aus und testen Sie Ihre Implementierung mittels der Klasse **`HeapSortRunner.java`**.

Hinweis: In der Klasse **`Heap.java`** kann es nützlich sein, wenn Sie sich zunächst Hilfsmethoden programmieren, um für den Index eines Elements im Array die Indexe von dessen linken und rechtem Kind im Heap zu berechnen (z.B. **`int getRightIndex(int index)`** / **`int getLeftIndex(int index)`**).

(14 Punkte)

Viel Spaß und viel Erfolg!