TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Lehrstuhl für Sprachen und Beschreibungsstrukturen
Grundlagen: Algorithmen und Datenstrukturen
Prof. Dr. Helmut Seidl, J. Kranz, A. Reuss, R. Vogler

SS 2016
Übungsblatt 7
29.05.2016

Abgabe: **05.06.2016** (bis 23:59 Uhr)

Aufgabe 7.1 (P) Binärer Heap

Führen Sie auf einem anfangs leeren Binären Heap folgende Operationen aus und stellen Sie die Zwischenergebnisse graphisch dar:

- build({15, 20, 9, 1, 11, 8, 4, 13, 17}),
- insert(7),
- delMin(),
- decreaseKey(20, 3),
- delete(8).

Aufgabe 7.2 (P) Quicksort

Die Laufzeit von Quicksort hängt von der Wahl des Pivotelements ab. Der Algorithmus aus der Vorlesung wählt immer das letzte Element als Pivotelement.

Wird in den folgenden Teilaufgaben ein anderes Element ausgewählt, vertauschen Sie dieses zunächst mit dem letzten Element und gehen Sie dann wie in der Vorlesung gezeigt vor.
Geben Sie jeweils ein Array der Länge 10 an, bei dem es besonders schlecht ist (d.h., es kommt zu vielen Vertauschungen) und ein Array der Länge 10, bei dem es besonders gut ist (wenige Vertauschungen), immer als Pivotelement

a) das erste Element zu wählen;

b) den Median aus dem ersten, letzten und mittleren Element zu wählen, d.h. bei einem Array A der Länge n den Median aus den Elementen $A[0], A[\lfloor (n-1)/2 \rfloor]$ und $A[n-1]$. Der Median ist der mittlere Wert: Z.B. ist es bei $[2, 1, 10]$ der Wert 2 (also nicht der Mittelwert). Ein zweielementiges Array wird direkt ohne Pivotelement sortiert, d.h. $[2, 1]$ wird durch eine Vertauschung zu $[1, 2]$.

Begründen Sie jeweils Ihre Antwort.

c) Sortieren Sie das Array $[5, 4, 6, 7, 3, 2, 8, 9, 1]$ mit Quicksort. Wählen Sie als Pivot-Element jeweils das letzte Element bzw. den Median **wie in Teil b)**.

Aufgabe 7.3 (P) Amore

Wir betrachten als Datenstruktur eine Liste L von Zahlen mit folgenden Operationen.

1. Die Methode L.`pushBack(`i`)`. Diese hängt die Zahl i an das Ende der Liste L.
2. Die Methode `findMinimum(unsigned int` m`)`:

```
1  int min := ∞
2  int counter := 0
3  while  L ist nicht leer  und  counter ≠ m  do
4  |  int nextElement := L.popFront() /* Entfernt das erste Element aus L und
   |   weist nextElement den Wert dieses Elements zu */
5  |  if nextElement < min then
6  |  |  min := nextElement
7  |  end
8  |  counter := counter + 1
9  end
10 return min
```

Jede Elementar-Operation (d.h. Zuweisung, Vergleich etc.), die innerhalb von `findMinimum` verwendet wird, soll konstante Laufzeit haben. Weiterhin soll innerhalb der Liste ein Zeiger auf das letzte Element vorhanden sein, so dass `pushBack` in konstanter Zeit ausgeführt wird.

(a) Ermitteln Sie die jeweilige Worst-Case-Laufzeit der Operationen `pushBack` und `findMinimum` bei einer Operationsfolge der Länge k. Berechnen Sie daraus eine pessimistische Abschätzung für die Worst-Case-Laufzeit von Operationsfolgen der Länge k.

(b) Zeigen Sie mithilfe einer amortisierten Analyse, dass die amortisierte Laufzeit der Operationen `pushBack` und `findMinimum` konstant (d.h. in $\mathcal{O}(1)$) ist, und bestimmen Sie die Laufzeit für Operationsfolgen der Länge k.

Aufgabe 7.4 (P) Zusatzaufgabe

In der Vorlesung lernen wir, Algorithmen aus theoretischer Sicht zu analysieren. Neben dieser Art der Algorithmen-Analyse gibt es die experimentelle Analyse. Hierbei testet man für Eingaben das Laufzeitverhalten eines Algorithmus.

In der Abbildung 1 sind für drei Algorithmen die gemessenen oberen Schranken für die Laufzeit t in Abhängigkeit der Eingabegröße n einer solchen experimentellen Analyse angegeben. Ordnen Sie die Algorithmen nach ihrer Laufzeit.

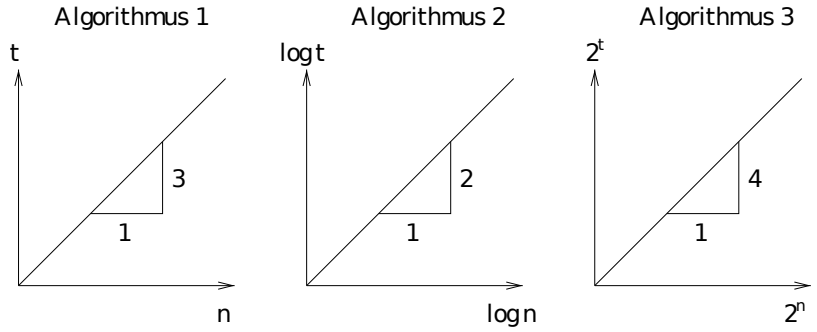

Abbildung 1: Graphen zu den Algorithmen in Aufgabe 1

Aufgabe 7.5 [7 Punkte] **(H) Slowsearch**

Es sei die folgende Implementierung („Slowsearch“) der binären Suche auf sortierten Feldern gegeben. Slowsearch kann lediglich mit Suchbereichen der Größe n mit $n = 2^k$ für ein $k \in \mathbb{N}_0$ arbeiten:

```
1  boolean slowsearch(int searchValue, int[] elements, int from, int n) {
2    if(n == 1) // Abbruch bei Feld der Länge 1
3      return elements[from] == searchValue;
4    assert(n % 2 == 0); // die Länge muss durch 2 teilbar sein
5    int middle = from + n/2; // erstes Element in oberem Teilfeld
6    if(searchValue < elements[middle]) {
7      for (int i = 0; i < middle; i++) // Debug-Ausgabe
8        System.out.println(elements[i]); // Debug-Ausgabe
9      return slowsearch(searchValue, elements, from, n/2);
10   } else
11     return slowsearch(searchValue, elements, middle, n/2);
12 }
```

Die `slowsearch(int searchValue, int[] elements, int from, int n)`-Methode sucht nach einem Wert `searchValue` im sortierten Feld `elements` im Bereich ab `from`, der eine Größe von `n` vielen Elementen aufweist. Um den Suchvorgang auf dem Terminal verfolgen zu können, gibt der Algorithmus (siehe Zeile 7 und 8) die untere Hälfte des Feldes (niedrigere Indices) aus, wenn die Suche in der unteren Hälfte fortgesetzt werden muss.

a) [3] Stellen Sie eine Rekursionsgleichung der Worst Case-Laufzeit von Slowsearch auf. Beachten Sie, dass die Debug-Ausgabe ein Teil des Algorithmus ist und daher in die Laufzeit einfließt. Nutzen Sie anschließend das Master-Theorem, um die Laufzeitklasse zu ermitteln.

b) [1] Beeinträchtigt die Debug-Ausgabe die Laufzeit entscheidend? Begründen Sie Ihre Antwort!

c) [3] Wir streichen nun die Debug-Ausgabe, löschen also Zeilen 7 und 8. Stellen Sie eine Rekursionsgleichung für die Laufzeit des resultierenden Algorithmus auf. Ist es auch hier möglich, mithilfe des vereinfachten Master-Theorems aus der Vorlesung eine Laufzeitklasse zu ermitteln? Geben Sie – mit oder ohne Nutzung des Master-Theorems – die Laufzeitklasse an und beweisen Sie die Korrektheit der Laufzeitklasse durch vollständige Induktion über die Größe des Eingabebereiches.

Aufgabe 7.6 [3 Punkte] **(H) Heapsort**

Führen Sie auf einem anfangs leeren Binären Heap folgende Operationen aus und stellen Sie die Zwischenergebnisse graphisch dar:

1. build($\{4, 2, 1, 9, 3\}$),
2. Sortierung der Zahlen durch Aufrufe von DeleteMinimum-Operationen.

Aufgabe 7.7 [5 Punkte] **(H) Programmieraufgabe: Radixchen**

In dieser Aufgabe geht es darum, das Sortierverfahren *RadixSort* für Arrays von Zahlen und Strings zu implementieren. Eine Vorlage für Ihre Implementierung finden Sie im Ordner `radixchen-angabe/`. Achten Sie darauf, die Signaturen der gegebenen Methoden nicht zu verändern; Sie dürfen das Projekt sonst beliebig erweitern. Gehen Sie vor wie folgt:

a) Implementieren Sie zunächst die Klasse `IntegerDescriptor`, die Informationen für RadixSort bezüglich der Sortierung von Integer-Zahlen zur Verfügung stellt. Ihre Implementierung muss dabei nur mit positiven Zahlen umgehen können; negative Zahlen als Element sollen zu einer Ausnahme führen. Beachten Sie bei Ihrer Implementierung die Kommentare der jeweiligen Methoden im Interface `KeyDescriptor`.

b) Implementieren Sie nun die Klasse `StringDescriptor`, die Informationen für RadixSort bezüglich der Sortierung von Strings zur Verfügung stellt. Ihre Implementierung muss dabei nur mit Strings umgehen können, die Buchstaben zwischen 'a' und 'z', 'A' und 'Z' bzw. '0' und '9' enthalten; alle übrigen Buchstaben als Teil von Strings, die als Element verwendet werden, sollen zu einer Ausnahme führen. **Tipp**: Ihre Implementierung benötigt Wissen über die Daten, die sortiert werden sollen. Implementieren Sie einen passenden Konstruktor.

c) Sie können nun mit der Implementierung des eigentlichen RadixSort-Algorithmus beginnen. Implementieren Sie dazu zunächst die Methode `void kSort(...)` der Klasse `RadixSort`.

d) Komplettieren Sie schließlich Ihre Implementierung durch die Methode `void sort(...)` der Klasse `RadixSort`.

Hinweis: Behandeln Sie Fehler in dieser Aufgabe sinnvoll. Sie können jeweils eine Ausnahme vom Typ `RuntimeException` werfen, die eine passende Fehlermeldung im Konstruktor erwartet.

Aufgabe 7.8 [5 Punkte] **(H) Programmieraufgabe: Binomilia**

Ein binomialer Haufen (*Binomial Heap*) besteht aus binomialen Bäumen. In dieser Aufgabe werden Sie zunächst die Klasse `BinomialTreeNode` für binomiale Bäume für Integer-Zahlen entwickeln, um diese anschließend zur Implementierung eines binomialen Haufens in der Klasse `BinomialHeap` zu nutzen. Eine Vorlage für Ihre Implementierung finden Sie im Ordner `binomilia-angabe/`. Achten Sie darauf, die Signaturen der gegebenen Methoden nicht zu verändern; sie dürfen das Projekt sonst beliebig erweitern. Implementieren Sie dabei die Klasse `BinomialTreeNode` wie folgt:

a) Implementieren Sie zunächst den Konstruktor der Klasse `BinomialTreeNode`; dieser Baut eine binomialen Baum vom Rang 0 aus einem Integer-Wert auf. Implementieren Sie anschließend die Methoden `int min()` und `int rank()`, die das minimale Element des Baumes mit dem aktuellen Knoten als Wurzel bzw. seinen Rang zurückgeben.

b) Implementieren Sie nun die Methode `BinomialTreeNode[] deleteMin()`, die aus dem Baum mit dem aktuellen Knoten als Wurzel das minimale Element entfernt. Die Methode liefert als Ergebnis ein Feld von Wurzeln von Bäumen zurück, in die der Baum durch das Entfernen des minimalen Elements zerfällt.

c) Implementieren Sie schließlich die Methode `BinomialTreeNode merge(BinomialTreeNode a, BinomialTreeNode b)`, die zwei an `a` und `b` gewurzelte Bäume vereint. Es sollen dabei keine neuen `BinomialTreeNode`-Objekt instaziert werden. Die Methode liefert eine Referenz auf diejenige der beiden Wurzeln zurück, an die der andere Baum angehängt wurde.

Nun können Sie mit der Implementierung der Klasse `BinomialHeap` beginnen:

a) Implementieren Sie zunächst den Konstruktor der Klasse `BinomialHeap` und die Methode `int min()`, die das minimale Element im binomialen Haufen zurückliefert.

b) Implementieren Sie nun die Methode `void insert(int value)`, die ein Element in den binomialen Haufen einfügt. **Tipp**: Implementieren Sie hierzu eine Methode `void merge(...)`, die eine Menge binomialer Bäume mit dem Haufen verschmilzt.

c) Implementieren Sie schließlich die Methode `int deleteMin()`, die das minimale Element aus dem Haufen löscht. **Tipp**: Nutzen Sie hier die Methode `void merge(...)` aus der vorherigen Teilaufgabe.

Hinweis: Behandeln Sie Fehler in dieser Aufgabe sinnvoll. Sie können jeweils eine Ausnahme vom Typ `RuntimeException` werfen, die eine passende Fehlermeldung im Konstruktor erwartet.