Elementare Algorithmen und Objektorientierte Programmierung

Übung 10

Auszuarbeiten bis 30.06.21

1. Hashtabelle (7 Punkte)

Realisieren Sie eine eigene generische Hashtabelle, die auf Wunsch der Verwenderin die Lineare Kollisionsstrategie, die Quadratische Kollisionsstrategie und Doppelhashing unterstützt.
Vermeiden Sie bei Ihrem Design Codeverdopplung und andere „unschöne“ Codierungsvarianten.

2. Zusammenfügen 2er sortierter, einfach-verketteter Listen zu einer neuen sortierten Liste (9 Punkte)

Entwickeln Sie eine Methode `Merge`, die zwei einfach-verkettete, aufsteigend-sortierte Listen *L1* und *L2* des Datentyps *List* zu einer neuen, aufsteigend-sortierten Liste *L3* zusammenfügt.

Die beiden Listen werden als Eingabeparameter an die Funktion übergeben. Die neue Liste wird als Rückgabewert zurückgegeben.

Achten Sie bei der Realisierung der Methode `Merge` auch auf Effizienz und Robustheit! Bestehende Knoten sollen wiederverwendet werden!

3. Laufzeitkomplexität (4 Punkte)

a) Was beschreibt die *O*-Notation? Welche Bedingung ist für die asymptotische Laufzeitkomplexität von essentieller Bedeutung?

b) Erklären Sie die nachfolgende mathematische Definition der O-Notation

$$g(n) = O(f(n))$$

$$O(f(n)) = \left\{ \begin{array}{l} g(n) : \exists c, n_0 > 0 \\ \ni 0 \le g(n) \le c\, f(n), \forall n \ge n_0 \end{array} \right\}$$

$$g(n) = O(f(n)) \Leftrightarrow \lim_{n \to \infty} \left| \frac{g(n)}{f(n)} \right| \le c$$

1 Hashtabelle

Lösungsidee
Mithilfe der doppelten hashing Methode eine Map (hashtable) implementieren. Der Hashwert wird bei dieser Methode mithilfe zweier verschiedenen hash funktionen generiert. Bei der ersten Hash-funktion **h()** rechnen wir die Position mit **key % size** aus. Bei der zweiten Hash-funktion **p()** wird eine alternative Position mit **key % (size – 1)** dannach **+ 1** ausgerechnet, somit ist sie unabhängig von der ersten Hash-funktion. Der finale Hashwert wird dann bei jeder Iteration mit **(h() + index * p()) % size** von der Map berechnet.
Bei der Lineraren und Quadratischen Kollisionsstrategie wird statt **p()** einfach entweder **i** oder **i^2** verwendet.

Code
Beiliegende .c und .h files.

Testfälle
siehe main.c file.

2 Zusammenfügen 2er sortierter, einfach-verketteter Listen zu einer neuen sortierten Liste

Lösungsidee
Die Klassen SingleLinkedList, Node und Merger selber implemetieren. Bei der Merger Klasse gibt es eine **mergeSortedLists** funktion mit der man zwei Listen zusammen fügen kann. Dabei wird beim Merge zuerst mit einer While über beide Listen drüber iteriert bis man bei einer am Ende angekommen ist. Dannach beide einzeln vom currentNode weiterlaufen lassen falls eine größer war als die andere. Beide Listen müssen natürlich gesortet sein.

Code
Beiliegendes MergeSortedLists java Projekt.

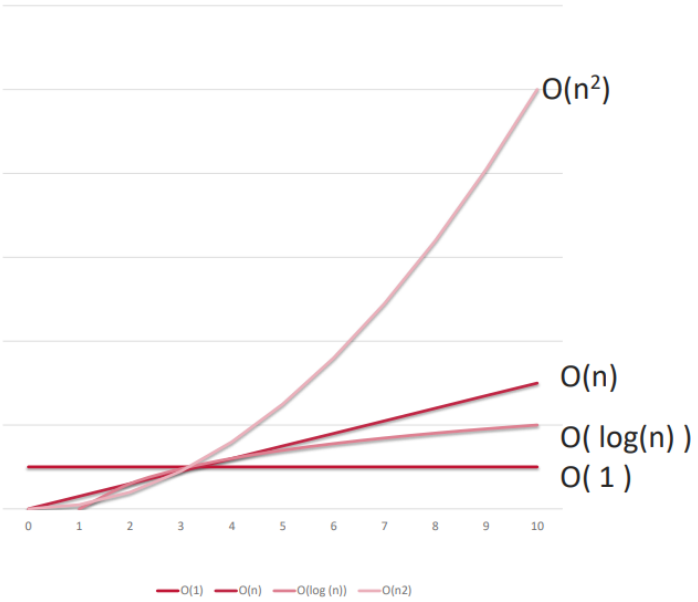
Testfälle
siehe test package.

3 Laufzeitkomplexität

a) Was beschreibt die O-Notation?

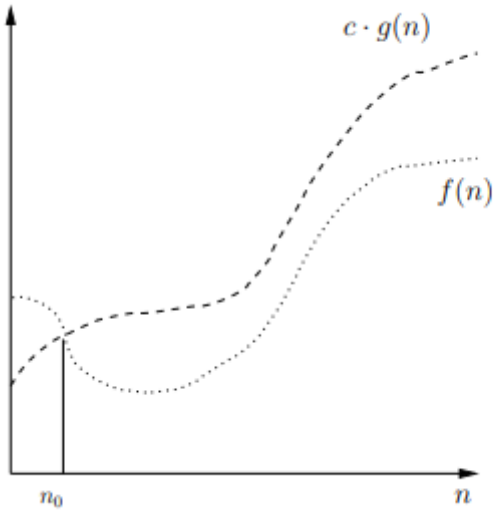
Die O-Notation beschhreibt den Wachstum der Laufzeit in abhängigkeit der Problemgröße. Dabei wird immer der schlechteste Fall betrachtet. Dazu gibt es verschiedene Komplexitätsklassen, die im folgendem Graph beschrieben sind. Auf der x Achse haben wir die Problemgröße und auf der y Achse die Laufzeit.

O(1)	Ideal
O(lg n)	Sehr gut
O(n)	Zufrieden stellend
$O(n^2)$	Unerwünscht
$O(a^n)$	Katastrophal



Welche Bedingung ist für die asymptotische Laufzeitkomplexität von essentieller Bedeutung?

Für die asysmtopische Laufzeitkomplexität ist es essentiell, dass diese erste ab einer grwissen Problemgröße gilt, da zuvor Konstate Operationen stärker ins Gewicht fallen können und somit kann es sein, dass bei einer kleineren Problemgröße ein Algorithmus mit O(log(n)) lansamer sein kann als ein Alorithmus mit O(n).



b) Erklären Sie die nachfolgende mathematische Definition der O-Notation

$$g(n) = O(f(n))$$

$$O(f(n)) = \left\{ \begin{array}{l} g(n) : \exists c, n_0 > 0 \\ \ni 0 \le g(n) \le c f(n), \forall n \ge n_0 \end{array} \right\}$$

$$g(n) = O(f(n)) \Leftrightarrow \lim_{n \to \infty} \left| \frac{g(n)}{f(n)} \right| \le c$$

Die Betrachtung der Laufzeit nur in Bezug auf eine obere Schranke.
Die Aussage gilt für alle n die >= n0 sind. Sprich erst bei einer gewissen Problemgröße.