Hochschule Darmstadt
Prof. Dr. Benjamin Meyer

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

3. Juni 2021

Programmieren, Algorithmen und Datenstrukturen II, SoSe 2021
Aufgabenblatt 4

4.1 Mergesort

Aus dem zweiten Aufgabenblatt ist Ihnen der Aufbau einer (doppelt verketteten) Liste bekannt. Erweitern Sie Ihre Implementierung der Liste um eine `merge`-Fuktion, die zwei sortierte Listen als Parameter bekommt (Call by Reference!) und eine neue ebenfalls sortierte Liste zurückgibt, welche alle Elemente der beiden ersten Listen enthält. Achten Sie darauf, nicht nur Referenzen auf die einzelnen Objekte zu speichern, sondern legen Sie echte (sog. tiefe) Kopien der ursprünglichen Listenobjekte an (Anmerkung: Beide Listen einfach hintereinander hängen und danach Ihre sort-Methode aufrufen zählt nicht!).
Mit dieser neuen Methode können Sie nun für die Liste eine weitere Methode schreiben: mergeSort(). Implementieren Sie den Mergesort-Algorithmus, wie aus der Vorlesung bekannt.

4.2 Heapsort

Ein alternativer Ansatz zum Sortieren der Liste ist der `Heapsort`. Hierfür wird die zu sortierende Liste (oder jedes andere Datenformat) als binärer, linksvollständiger Baum interpretiert.
Stellen Sie als Vorbereitungsschritt in der neuen Methode `printTree()` Ihre Liste daher auf der Konsole visuell als Baum dar. Eine mögliche Darstellung ist in Listing 1 gegeben – wobei Sie natürlich den Inhalt des Listenelements anzeigen sollen statt dem Zugriffsoperator. Gehen Sie für die Darstellung der Einfachheit halber von Listen mit max. 15 Elementen (also 4 Ebenen) aus. Implementieren Sie zudem Hilfsfunktionen, um von einem Index in der Liste auf dessen Vaterknoten sowie linken und rechten Kindsknoten zugreifen zu können (falls diese vorhanden sind): `getParent(int index)`, `getLeftChild(int intex)`, `getRightChild(int index)`.

```
           ___________L[0]___________
          /                          \
      ____L[1]____                ____L[2]____
     /            \              /            \
  _L[3]_        _L[4]_        _L[5]_        _L[6]_
 /      \      /      \      /      \      /      \
L[7]    L[8]  L[9]   L[10]  L[11]   L[12] L[13]   L[14]
```

Listing 1: Konsolendarstellung eines Baumes auf Basis der Liste L.

Diese Baumstruktur kann anschließend genutzt werden, um den aus der Vorlesung bekannten Heapsort zu implementieren. Schreiben Sie hierfür die Listen-Methode `heapSort()`. Geben Sie zu Debugging-Zwecken nach jedem Sortierschritt den (Teil-)Baum neu aus.

4.3 Quicksort

Für den Quicksort als letzten Sortieralgorithmus müssen Sie ein sog. Pivot-Element wählen und zwei Teillisten erzeugen, die alle kleineren bzw. alle größeren Elemente der ursprünglichen Liste enthalten. Um den in der Vorlesung diskutierten Worst Case möglichst zu vermeiden, bietet es sich an, das Pivot-Element jeweils als zufälliges Element der Liste zu wählen (und nicht immer das erste oder letzte Listenelement zu nehmen).
Versuchen Sie, den Algorithmus „in place“ zu implementieren, erzeugen Sie also keine neue Liste. Recherchieren Sie stattdessen, wie genau beim Quicksort die Elemente innerhalb der Liste getauscht werden.
(Anmerkung: da das gewählte Pivot-Element selber immer zwischengespeichert werden muss, wird Quicksort in einigen Quellen nicht als in-place beschrieben, sondern benötigt einen zusätzlichen Speicherbedarf von O(log(n)), das ist für die hier gewünschte Standard-Implementierung aber in Ordnung).

4.4 Laufzeit

Testen Sie die Laufzeit der verschiedenen Sortier-Algorithmen! Erzeugen Sie hierfür jeweils eine Liste mit 100.000, 200.00, 300.00, 400.000 und 500.000 zufälligen Elementen und wenden Sie die unterschiedlichen Sortier-Algorithmen an. Speichern Sie sich die Zeit vor Aufruf der jeweiligen Methode und danach. Eine sekundengenaue Messung soll uns hier reichen, Sie können hierfür die Methode `time(NULL)` nutzen (vorher `<ctime>` inkludieren!), um die Systemzeit in Sekunden seit dem 01.01.1970 zu erhalten. Vergleichen Sie die Ergebnisse.
Berechnen Sie zudem die theoretische Laufzeit der verschiedenen Algorithmen mittels des Master-Theorems. Passen die gemessenen Zahlen zur Theorie?

4.5 UnitTests

Da bei dem letzten Aufgabenblatt noch viele Fragen bei den QUnitTests offen waren (und die Einrichtungs-Anweisungen der Vorlesung vielleicht nicht ganz so klar waren wie erhofft), erneut die Aufforderung: Schreiben Sie UnitTests, um die korrekte Funktionsweise Ihrer Sortier-Algorithmen sicherzustellen.
Genauer: Erzeugen Sie eine Liste mit mind. 5 zufällig erstellten Elementen (entweder hardcoded oder per `rand()`) und schreiben Sie einen Test, der sicherstellt, dass nach Aufruf des jeweiligen Sortieralgorithmus' alle Elemente der Liste in sortierter Reihenfolge stehen.

4.6 Stabiler Quicksort

Wie in der Vorlesung besprochen ist der Quicksort in der nativen Implementierung nicht stabil (d.h. gleiche Elemente können die Position tauschen). Wenn wir bereit sind, dem Algorithmus einen zusätzlichen Speicherplatz von O(n) zu gewähren (also neue Listen erzeugt werden), lässt sich die Stabilität jedoch einhalten. Versuchen Sie, Ihre Implementierung des Quicksorts so anzupassen, dass gleiche Elemente nicht mehr die Plätze tauschen (wie immer gibt es auch hierfür schon Internetquellen mit Beispielcode – aber können Sie die Aufgabe auch eigenständig lösen?).