Theorie

1. Doppelt verkettete Listen

Im Unterricht haben Sie gesehen, dass Listen, welche auf einem Elementtyp mit einer `Object`-Referenz basieren, für beliebige Objekte verwendet werden können. Diese maximale Verallgemeinerung des Listeninhalts hat aber auch Nachteile: es kann nicht davon ausgegangen werden, dass alle Objekte eine gewünschte Methode zur Verfügung stellen (z.B. `print()`), mit Ausnahme derer, die bereits in der Klasse `Object` vordefiniert wurden. Bei der Methode `print()` finden wir Abhilfe durch die Verwendung der Methode `toString()`. Dieser direkte Ersatz ist aber nicht immer möglich. Stellen Sie sich zum Beispiel vor, dass die Datenobjekte untereinander verglichen werden sollten, um sie beispielsweise sortieren zu können. In einem solchen Fall ist es sehr hilfreich, wenn alle Datenobjekte eine Vergleichsmethode besitzen, um zwei Datenobjekte miteinander vergleichen zu können. Genau für diesen Zweck gibt es das Interface `Comparable`. Alle Klassen, die das Interface `Comparable` implementieren, müssen die Methode `compareTo(…)` anbieten. Diese Methode entscheidet, ob zwei Objekte als gleichwertig (= 0) betrachtet werden, oder welches von den beiden zuerst aufgeführt werden soll (-1 bzw. 1).

2. Merge-Sort

Das Sortierverfahren Merge-Sort (genauer: der 2-Wege Merge-Sort) eignet sich besonders für Datenstrukturen mit ausgeprägt sequentiellem Charakter, wie beispielsweise ein File-Stream oder eine lineare Liste, da deren Elemente durch den Algorithmus immer schön der Reihe nach bearbeitet werden.

Die im Unterricht vorgestellte *iterative* Sortierung auf Arrays kann in gewisser Weise als Umkehrung des *Quick-Sort*, den Sie in Algorithmen & Datenstrukturen 1 kennengelernt hatten, aufgefasst werden: Im Quick-Sort wurde ein Array in zwei Teile zerlegt, so dass der rechte die grösseren Elementen, der linke die kleineren Elemente bekam, wobei das grösste Element linken Teils kleiner oder gleich dem kleinsten Element des rechten Teils war. Dieses Verfahren wurde mit beiden Teilen rekursiv fortgesetzt, bis der ganze Array sortiert war.

Der Schwachpunkt dieses Verfahrens bestand in der Schwierigkeit, die beiden Teile möglichst gleich gross hinzubekommen.

Im *Merge-Sort* gehen wir genau umgekehrt vor: Zunächst wird der Array als N Mengen einzelner Elemente aufgefasst. Sie werden paarweise zu N/2 Zweier-Gruppen zusammengemischt, dann werden benachbarte Zweiergruppen zu $N/4$ Vierergruppen gemischt. Nach jedem Mischdurchgang durch den ganzen Array verdoppelt sich die Gruppengrösse, bis wir nach etwa $log_2(N)$ Mischdurchgängen den ganzen Array sortiert haben. Da wir für jeden Mischdurchgang insgesamt N Vergleiche benötigen, beläuft sich der Sortieraufwand auf $O(N \cdot log\ N)$.

Im Gegensatz zum *Quick-Sort* können wir allerdings diese Performance garantieren! Anderseits benötigen wir im *Merge-Sort* einen zweiten Array der Grösse N als Zwischenspeicher für Mischoperationen. Beim *Merge-Sort* hat sich lediglich die Unterscheidung einiger Spezialfälle am Ende des Arrays als etwas unbequem herausgestellt (verkürzte Gruppen).

Man kann den *Merge-Sort* aber auch *rekursiv* implementieren, was besonders mit verketteten Listen zu einer schönen und einfachen Lösung führt, die wir hier in Pseudocode vorstellen:

```
void mergeSort(Liste a){
  zerlege a in möglichst zwei gleich grosse Teile a1 und a2.
  sortiere a1.
  sortiere a2.
  mische a1 und a2 zusammen -> a.
}
```

Für das Mischen zweier Arrays der Grössen N und M benötigt man einen Hilfsarray der Grösse $N+M$, weil eine Mischung *in-place* (d.h. ohne zusätzliche Datenstruktur) untragbar teuer ist (das Verschieben von Teilen eines Arrays erfordert viele Kopiervorgänge). Mit verketteten Listen ist *in-place-merging* einfacher, da man mit geringem Aufwand an beliebiger Stelle Elemente einfügen oder entfernen kann.