Gottfried Wilhelm Leibniz Universität Hannover
Institut für Theoretische Informatik

Eine GUI zur Visualisierung von Sortierverfahren

Bachelorarbeit

Jan Strothmann
Matrikelnr. 3133690

Hannover, den 7. Juni 2024

Erstprüfer:	PD Dr. Arne Meier
Zweitprüfer:	Prof. Dr. Heribert Vollmer
Betreuerin:	M. Sc. Vivian Holzapfel

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 7. Juni 2024

____

Jan Strothmann

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **2**

2.1 *Selectionsort* 3
2.2 *Insertionsort* 4
2.3 *Mergesort* 5
2.4 *Quicksort* 6

3 Darstellung **9**

4 Software **12**

4.1 Wahl der Bibliothek für die graphische Benutzeroberfläche 14
4.2 Modularität 14
4.2.1 Hauptklasse 15
4.2.2 Darstellung der Liste 16
4.2.3 Algorithmen 20
4.2.4 Darstellung des Aufrufstapels 21
4.2.5 Einstellungen 22
4.2.6 Bearbeiten der Liste durch den Benutzer 23
4.2.7 Speichern und Laden von Zuständen 23
4.2.8 Zurück-Funktion 24
4.3 Größenbeschränkungen 26

5 Zusammenfassung und Ausblick **28**

1 Einleitung

Ein Themenpunkt der Veranstaltung „Datenstrukturen und Algorithmen“ der Leibniz Universität Hannover ist das Sortieren von Listen. Dort werden unter anderem die Sortierverfahren *Selectionsort*, *Insertionsort*, *Mergesort* und *Quicksort* vorgestellt. Das Ziel dieser Bachelorarbeit ist es, eine Software zur schrittweisen Darstellung dieser Sortierverfahren für die Vorlesung zu entwickeln. Die Software soll die Sortierverfahren an einem vom Benutzer frei wählbaren Beispiel Schritt für Schritt ausführen und darstellen. Der aktuelle Zustand der Liste und die Variablen des Sortierverfahrens werden graphisch dargestellt. So lassen sich die Verfahren relativ leicht im Detail nachvollziehen, ohne zusätzlichen kognitiven Aufwand beispielsweise für die Berechnung von numerischen Indizes zu benötigen. Gleichzeitig wird der Pseudocode angezeigt, und die gerade ausgeführte Stelle hervorgehoben. Die Software ist im Wesentlichen in Python implementiert, und erzeugt eine graphische Benutzeroberfläche. Pseudocode aus dem Skript der Vorlesung im Wintersemester 23/24 wird, soweit verfügbar, verwendet. Für *Quicksort* wird im Skript *Python*-, aber kein Pseudocode verwendet. Das Skript ist für Angehörige der Leibniz Universität Hannover über Stud.IP verfügbar.

Um bestimmte Stellen in der Ausführung der Algorithmen zu zeigen, ohne sie lange suchen zu müssen, lässt sich der aktuelle Zustand als Datei speichern und laden, so dass während der Vorlesung schnell auf zuvor gewählte Stellen der Algorithmen zugegriffen werden kann. Um Fragen der Studierenden schnell am Beispiel beantworten zu können, ist eine schrittweise Zurück-Funktion implementiert worden. Dazu wurde das Speichern vergangener Zustände im Hauptspeicher einer Rückwärts-Implementierung der Algorithmen vorgezogen. Damit im Algorithmus sehr oft wiederholte Schritte nicht vollständig schrittweise durchlaufen werden müssen, kann mit einer Überspringen-Funktion zum nächsten Ende einer Schleife oder zum Beginn der Funktionen `Merge` und `Partition` gesprungen werden. Es ist dem Benutzer möglich die aktuelle Darstellung als Bilddatei zu exportieren, zur Verwendung in Präsentationsfolien, auf Webseiten oder in anderen Medien.

Diese Arbeit beschreibt zunächst die anfangs genannten Sortierverfahren, dann wird die gewählte Darstellung vorgestellt und anschließend die im Rahmen der Arbeit implementierte Software näher beschrieben. Zum Schluss folgen eine Zusammenfassung und ein kurzer Ausblick auf mögliche Erweiterungen der Software.

2 Grundlagen

Die Sortierverfahren *Selectionsort*, *Insertionsort*, *Mergesort* und *Quicksort* werden von der Software dargestellt, und in diesem Kapitel näher beschrieben. Die Auswahl der Verfahren ist wegen des Inhalts der Vorlesung erfolgt und durch die Aufgabenstellung festgelegt. Diese Verfahren werden in der Vorlesung detailliert erklärt. Um die Verfahren genauer zu beschreiben werden zunächst einige Begriffe definiert:

(1) Eine *Liste* ist eine Sammlung von Elementen, die einen gemeinsamen Typ haben. Elemente haben in einer Liste eine Reihenfolge, und können auch mehrfach vorkommen. Entsprechend dieser Reihenfolge hat das erste Element der Liste den Index 0, das zweite Element den Index 1, etc. Elemente mit kleinerem Index werden als *links*, Elemente mit größerem Index als *rechts* eines Elements bezeichnet, dies entspricht auch der Position der Elemente in der Darstellung der Liste (siehe Kapitel 3). *Nachbarelemente* eines Elements befinden sich in der Reihenfolge der Liste direkt vor oder nach diesem Element. Der *Anfang* oder *Beginn* der Liste ist das erste Element der Reihenfolge, das *Ende* der Liste ist das letzte Element der Reihenfolge. Eine *Teilliste* ist eine Sammlung in der Reihenfolge zusammenhängender Elemente einer Liste, die selbst als Liste betrachtet werden kann. Änderungen an einer Teilliste werden auch auf die Gesamtliste angewandt.

Definition 1 *Eine* Liste *ist eine Sammlung von $n \in \mathbb{N}_0$ Elementen $a_0, a_1, \dots, a_{n-1}$. Der* Anfang *oder* Beginn *der Liste ist das Element a_0, das* Ende *der Liste ist das Element a_{n-1}. Ein Element a_i ist* links *eines Elements a_j, falls $i < j$, und* rechts *des Elements falls $i > j$. Die Elemente a_{i-1} und a_{i+1} sind* Nachbarelemente *des Elements a_i. Eine* Teilliste *der Elemente $l \in \mathbb{N}_0$ bis $r \in \mathbb{N}_0$, $0 \leq l \leq r < n$, einer Liste ist eine Sammlung aufeinander folgender Elemente $a_l, a_{l+1}, \dots, a_{r-1}, a_r$ der Liste.*

(2) **Definition 2** *Eine Liste ist nach einer Ordnung oder Halbordnung $\prec$-*sortiert*, wenn für alle $i, j \in \{0, 1, \dots, n-1\}$ gilt $i < j \implies a_i \prec a_j$. Leere Listen und Listen mit nur einem Element gelten als* sortiert*, unabhängig von $\prec$.*

(3) Ein *Sortierverfahren* ist ein Algorithmus, der die Reihenfolge einer Liste verändert, so dass sie nach einer Ordnung oder Halbordnung sortiert ist, ohne die Werte der Elemente der Liste zu verändern.

Zur Veranschaulichung arbeiten alle Algorithmen hier mit einer als Array implementierten Liste natürlicher Zahlen als Eingabe, das sie aufsteigend nach dem Wert der Zahl sortieren. Es gibt keinen Rückgabewert, das Array wird *in-place* sortiert. Der Pseudocode der nun beschriebenen Sortierverfahren stammt aus dem Skript der Vorlesung „Datenstrukturen und Algorithmen" im Wintersemester 23/24.

2.1 *Selectionsort*

Das Sortierverfahren *Selectionsort* wählt jeweils das nächste Element der Liste aus den noch nicht sortierten Elementen, und bringt es durch Vertauschen von zwei Elementen an die der Ordnung entsprechenden Stelle [Knu98, S. 138, 139]. Im Folgenden wird *Straight selection sort* [Knu98, S. 139 Algorithm S] aus dem Standardwerk von Knuth beschrieben, mit einer Änderung: Statt am Ende der Liste zu beginnen, und das größte Element auszuwählen, beginnt diese Implementierung des Algorithmus am Beginn der Liste und wählt das kleinste Element. Knuth stellt das gundlegende Verfahren zunächst auf diese Weise vor [Knu98, S. 138], implementiert den Algorithmus aber am Ende der Liste beginnend und das größte Element wählend [Knu98, S. 139 Algorithm S].

Für das erste bis vorletzte Element i der Liste (Algorithmus 1 Zeile 2) wird nacheinander jeweils das kleinste der noch nicht sortierten Elemente gesucht. Dazu wird zunächst eine Referenz auf i gespeichert (Algorithmus 1 Zeile 3). Diese wird nun immer ersetzt, wenn ein kleineres Element gefunden wird. Für alle Elemente j rechts des Elements i (Algorithmus 1 Zeile 4) wird nacheinander geprüft, ob ihr Wert kleiner ist als der des Elementes der gespeicherten Referenz (Algorithmus 1 Zeile 5). Ist das der Fall, wird die gespeicherte Referenz mit einer Referenz des Elements j überschrieben (Algorithmus 1 Zeile 5). Nachdem so für alle Elemente j verfahren wurde, zeigt die Referenz auf das kleinste Element von dem Element i aus bis zum Ende der Liste. Dieses Element wird mit Element i vertauscht (Algorithmus 1 Zeile 6), es befindet sich nun an seiner Stelle der sortierten Liste.

Algorithmus 1 : *Selectionsort*

```
1 Selectionsort(Array A von natürlichen Zahlen):
2     for i ← 0 bis n − 2 do
3         minimum ← i
4         for j ← i + 1 bis n − 1 do
5             if A[j] < A[minimum] then minimum ← j
6         Vertausche A[i] mit A[minimum]
```

2.2 Insertionsort

Das Sortierverfahren *Insertionsort* fügt jedes Element der Liste nacheinander an der der Ordnung entsprechenden Stelle in eine bereits sortierte linke Teilliste ein [Knu98, S. 80]. *Straight insertion sort* aus dem Standardwerk von Knuth wird nun beschrieben [Knu98, S. 80, 81 Algorithm S]. Auf einen Unterschied zum in der Vorlesung verwendeten Pseudocode (Algorithmus 2) wird am Ende des Abschnitts eingegangen.

Die Elemente der Liste werden nacheinander vom zweiten bis zum letzten Element der Liste wie folgt behandelt (Algorithmus 2 beginnt beim ersten Element der Liste, siehe Zeile 2): Das Element wird in einer Variable gespeichert, da der Platz des Elements in der Liste durch diesen Algorithmus von einem anderen Element besetzt werden könnte (Algorithmus 2 Zeile 3 erste Zuweisung). Zusätzlich wird eine Referenz auf ein Element der Liste gespeichert, zunächst auf den linken Nachbarn des Elements (Algorithmus 2 Zeile 3 zweite Zuweisung).

Solange die Referenz auf ein Element der Liste verweist, und das Element der Referenz größer als das Element ist, wird das Element der Referenz eine Stelle weiter rechts in der Liste gespeichert, und die Referenz um ein Element nach links verschoben (Algorithmus 2 Zeile 4). Wie bereits erwähnt wird das ursprüngliche Element dabei überschrieben, deswegen wurde es zuvor in der Variable gespeichert. Das Element, mit dem es überschrieben wurde, ist in der Liste nun doppelt vorhanden. Es wird durch den Algorithmus aber noch überschrieben, entweder durch das Element links davon, oder durch das Element aus der Variable, je nach dem ob die Bedingung zu Beginn des Absatzes nach Veränderung der Referenz erneut erfüllt ist. Die Bedingung wird nun erneut geprüft.

Wenn die Referenz auf kein Element der Liste verweist, oder das Element der Referenz kleiner als oder gleich dem Element ist, wird das Element aus der Variable eine Stelle rechts der Referenz in der Liste gespeichert (Algorithmus 2 Zeile 5). Dann wird das nächste Element behandelt.

Der von Knuth vorgestellte Algorithmus [Knu98, S. 80, 81 Algorithm S] beginnt mit dem zweiten Element, der in der Vorlesung vorgestellte Algorithmus mit dem ersten Element der Liste (Algorithmus 2 Zeile 2). Dieses erste Element wird in einer Variable gespeichert (Algorithmus 2 Zeile 3 erste Zuweisung). Die Bedingung der inneren Schleife (Algorithmus 2 Zeile 4) ist für das erste Element der Liste immer falsch: Der Index i des ersten Elements ist 0. Index $j \leftarrow i - 1$ ist daher -1. Also ist $j \ngeq 0$, die Konjunktion beider Teilbedingungen ist daher falsch. Algorithmus 2 speichert nun das Element aus der Variable, also das erste Element der Liste, am Index $-1 + 1 = 0$, genau am Beginn der Liste. Die Liste wurde also nicht verändert. Variablen aus der letzten Iteration werden in der nächsten Iteration der äußeren Schleife (Algorithmus 2 Zeile 2) nicht verwendet. Von hier an verhält sich der Algorithmus der Vorlesung so wie der von Knuth vorgestellte Algorithmus.

Algorithmus 2 : *Insertionsort*

```
1 Insertionsort(Array A von natürlichen Zahlen):
2     for i ← 0 bis n − 1 do
3         key← A[i], j ← i − 1
4         while j ≥ 0 und A[j] > key do A[j + 1] ← A[j], j ← j − 1
5         A[j + 1] ← key
```

2.3 *Mergesort*

Das Sortierverfahren *Mergesort* fügt sortierte Teillisten unter Beachtung der Ordnung zu einer Liste zusammen [Knu98, S. 158]. Das kleinste Element beider Teillisten lässt sich durch einen Vergleich der ersten Elemente der Teillisten leicht finden [Knu98, S. 158]. Das in der Vorlesung vorgestellte Verfahren teilt eine Liste rekursiv in jeweils zwei möglichst gleich lange Teillisten, bis diese nur ein Element enthalten, und fügt diese anschließend unter Beachtung der Ordnung wieder zusammen. Dieses Verfahren wird nun genauer beschrieben, es beginnt mit einem Aufruf der *Mergesort*-Funktion mit der Liste und den Indizes des ersten und letzten Elements der Liste.

Die *Mergesort*-Funktion bekommt eine Liste A und zwei Referenzen l und r auf Elemente der Liste übergeben. Dabei ist l die linke und r die rechte Grenze der zu sortierenden Teilliste.

Falls $l < r$ ist, wird der Wert des Ausdrucks $\lfloor (l + r)/2 \rfloor$ in einer Variable m gespeichert (Algorithmus 3 Zeile 3). An diesem Index wird die Liste in zwei Teillisten geteilt. Die *Mergesort*-Funktion wird auf beiden Teillisten aufgerufen, zunächst mit A, l und m, und dann mit $A, m + 1$ und r (Algorithmus 3 Zeile 4, 5). Nun sind beide Teillisten sortiert. Sie werden durch Aufruf der *Merge*-Funktion mit A, l, m und r unter Beachtung der Ordnung zusammengefügt (Algorithmus 3 Zeile 6).

Die *Merge*-Funktion bekommt als Argumente eine Liste A und drei Referenzen l, m und r übergeben. Die Liste wird von diesen Referenzen in zwei Teillisten unterteilt, eine Teilliste von l bis m und eine Teilliste von $m + 1$ bis r. Diese Teillisten werden nun in einer Liste B zusammengefügt, die anschließend nach A kopiert wird. Beide Teillisten werden jeweils von einer Referenz i und j durchlaufen, i ist zunächst l und j ist zunächst $m + 1$ (Algorithmus 3 Zeile 8). Die Referenzen i und j zeigen immer auf das kleinste Element ihrer Teilliste, das noch nicht nach B kopiert wurde, und werden daher am Anfang ihrer Teilliste initialisiert. Immer, wenn ein Element nach B kopiert wird, wird auch die entsprechende Referenz auf ihren rechten Nachbarn, also das nächst-kleinste Element der Teilliste verschoben. Eine Referenz k speichert die Stelle in Liste B an die kopiert wird, zunächst l (Algorithmus 3 Zeile 8).

Solange das Ende einer Teilliste noch nicht erreicht wurde (Algorithmus 3 Zeile 9), wird das kleinere Element der Referenzen i und j nach k kopiert, und wie erwähnt die jeweilige

Referenz um ein Element nach rechts verschoben (Algorithmus 3 Zeile 10). Außerdem wird k um ein Element nach rechts verschoben. Dann wird erneut geprüft, ob eine der Referenzen i und j das Ende ihrer Teilliste erreicht hat.

Ist das Ende einer Teilliste erreicht, werden die restlichen Elemente der anderen Teilliste nach B an die Stelle k und folgende kopiert (Algorithmus 3 Zeile 12 bis 15). Die Liste B von l bis r ist nun eine vollständig sortierte Liste der Elemente beider Teillisten.

Zum Schluss wird der Bereich von l bis r aus B nach A kopiert (Algorithmus 3 Zeile 16).

Algorithmus 3 : *Mergesort*

```
1  Mergesort(Array A von natürlichen Zahlen, Indizes l, r):
2      if l < r then
3          m ← ⌊(l + r)/2⌋
4          Mergesort (A, l, m)
5          Mergesort (A, m + 1, r)
6          Merge (A, l, m, r)
7  Merge(Array A von natürlichen Zahlen, Indizes l, m, r):
8      i ← l, j ← m + 1, k ← l
9      while i ≤ m and j ≤ r do   // so lange wie beide noch Elemente haben
10         if A[i] ≤ A[j] then B[k] ← A[i], i ← i + 1, else B[k] ← A[j], j ← j + 1
11         k ← k + 1
12     if i > m then          // alle Elemente der linken Teilliste kopiert
13         for h ← j bis r do B[k] ← A[h], k ← k + 1
14     else                   // alle Elemente der rechten Teilliste kopiert
15         for h ← i bis m do B[k] ← A[h], k ← k + 1
16     for h ← l bis r do A[h] ← B[h]        // Ergebnis von B nach A kopieren
```

2.4 *Quicksort*

Das Sortierverfahren *Quicksort* wählt ein Element und platziert dieses an seiner endgültigen Stelle entsprechend der Ordnung [Knu98, S. 113]. Das gewählte Element wird als Pivotelement bezeichnet. Während nach der Stelle gesucht wird, werden die anderen Elemente grob geordnet: Elemente kleiner als das gewählte Element werden links davon, Elemente größer als das gewählte rechts davon platziert [Knu98, S. 113]. Das Verfahren wird dann für die Bereiche links und rechts des gewählten Elements jeweils wiederholt, bis die Liste vollständig sortiert ist [Knu98, S. 114]. Im folgenden Abschnitt wird zunächst die *Quicksort*-Funktion von Hoare [Hoa61b; Hoa62] beschrieben. Die von Hoare vorgestellte *Partition*-Funktion [Hoa61a; Hoa62] arbeitet anders als die in der Vorlesung verwendete *Partition*-Funktion und wird hier nicht näher betrachtet. Beide Funktionen wählen verschiedene Pivotelemente, und arbeiten unterschiedlich, bringen aber beide ihr Pivotelement an seine der Ordnung entsprechenden

Stelle, und ordnen die Liste in Elemente kleiner als das Pivotelement, links des Pivotelements, und Elemente größer als das Pivotelement, rechts des Pivotelements. Die *Partition*-Funktion der Vorlesung wird im Anschluss beschrieben.

Die *Quicksort*-Funktion bekommt als Argumente die Liste A und zwei Referenzen auf Elemente der Liste l und r übergeben (Algorithmus 4 Zeile 1). Diese begrenzen die betrachtete Teilliste, l zeigt auf den Anfang der Teilliste, r auf das Ende. Falls $l < r$ ist (Algorithmus 4 Zeile 2), wird die *Partition*-Funktion mit A, l und r aufgerufen (Algorithmus 4 Zeile 3). Die in der Vorlesung vorgestellte Partition-Funktion gibt eine Referenz auf das Pivotelement zurück, die Partition-Funktion von Hoare [Hoa61a] gibt Referenzen auf die Elemente links und rechts des Pivotelements zurück. Anschließend wird rekursiv mit den Teillisten links und rechts des Pivotelements verfahren. Die *Quicksort*-Funktion wird für die linke Teilliste mit A, l und einer Referenz auf das Element links des Pivotelements aufgerufen (Algorithmus 4 Zeile 4), und für die rechte Teilliste mit A, einer Referenz auf das Element rechts des Pivotelements und r (Algorithmus 4 Zeile 5).

Die *Partition*-Funktion bekommt als Argumente die Liste A und zwei Referenzen auf Elemente der Liste l und r übergeben (Algorithmus 4 Zeile 6). Diese begrenzen die betrachtete Teilliste, l zeigt auf den Anfang der Teilliste, r auf das Ende. Das Pivotelement ist in dieser Implementierung immer das letzte Element der Teilliste, das Element der Referenz r. Außerdem wird eine Referenz i gespeichert, die zunächst auf $l - 1$ zeigt (Algorithmus 4 Zeile 8). Diese Referenz gibt den zukünftigen Index des Pivotelements an, sie wird immer dann verändert, wenn ein Element kleiner als das Pivotelement ist. Für alle Elemente j der Teilliste bis auf das letzte (Algorithmus 4 Zeile 9) wird nacheinander geprüft, ob ihr Wert kleiner ist als der Wert des Pivotelements (Algorithmus 4 Zeile 10). Ist das der Fall, wird die Referenz i um ein Element nach rechts verschoben (Algorithmus 4 Zeile 11), und die Elemente der Referenzen i und j vertauscht (Algorithmus 4 Zeile 12). Nachdem für alle Elemente j so verfahren wurde, wird das Pivotelement mit dem Element der Referenz $i + 1$ vertauscht (Algorithmus 4 Zeile 13). Die Referenz $i + 1$ wird von der *Partition*-Funktion an die aufrufende Funktion zurückgegeben (Algorithmus 4 Zeile 14). Sie zeigt auf das Pivotelement dieser Inkarnation, und wird von der *Quicksort*-Funktion für die Aufteilung der Liste in Teillisten verwendet.

Algorithmus 4 : *Quicksort*

```
1  Quicksort(Array A von natürlichen Zahlen, Indizes l, r):
2      if l < r then
3          pivot←Partition(A, l, r)
4          Quicksort(A, l, pivot − 1)
5          Quicksort(A, pivot + 1, r)
6  Partition(Array A von natürlichen Zahlen, Indizes l, r):
7      pivot← A[r]                 // wähle letztes Element als Pivotelement
8      i ← l − 1                                       // temporärer Pivotindex
9      for j ← l bis r − 1 do                        // alle Elemente außer Pivot
10         if A[j] ≤ pivot then
11             i ← i + 1          // verschiebe temp. Pivotindex nach rechts
12             Vertausche A[i] mit A[j]       // kleinere Elemente nach links
13     Vertausche A[i + 1] mit A[r] // bringe Pivotelement an seine Position
14     return i + 1    // Position des Pivotelements, das die Liste teilt
```

3 Darstellung

In diesem Kapitel wird die gewählte Darstellung der Liste und Variablen der Sortierverfahren in der Software näher beschrieben. Die Elemente der Liste werden jeweils als Rechteck dargestellt, der Hintergrund farbig markiert, der Rand umrissen und der Wert darauf geschrieben. Das erste Element der Liste ist links, die folgenden Elemente sind jeweils direkt rechts neben ihrem Nachbarn. Werden mehrere Listen gleichzeitig dargestellt, befinden sie sich übereinander. Elemente, die durch den Algorithmus bereits sortiert wurden, also entweder in einer sortierten Teilliste geordnet sind, oder die bereits an ihrer finalen Position sind, werden von bisher unsortierten Elementen farblich unterschieden. Diese Unterscheidung ist aus dem Skript der Vorlesung übernommen. Welche Farbe für sortierte und unsortierte Elemente verwendet wird, ist in einer Legende am oberen linken Rand der Darstellung notiert. Für *Mergesort* werden Teillisten durch einen horizontalen Abstand zwischen den Elementen der einzelnen Teillisten dargestellt. *Quicksort* verwendet diese Darstellung nicht, weil die Teillisten vom Pivotelement farblich bereits ausreichend getrennt sind.

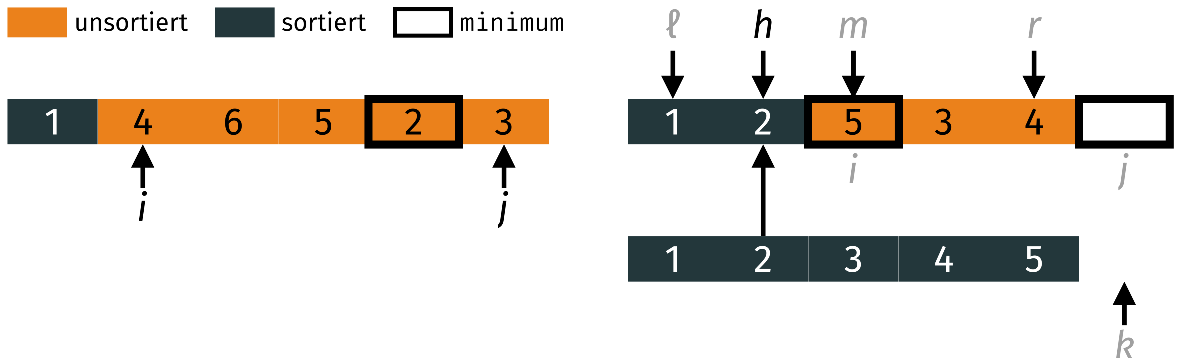
Referenzen auf Elemente der Liste werden auf zwei verschiedene Arten dargestellt, je nach ihrer Bedeutung für den Algorithmus:

(1) Referenzen, die gespeichert werden, weil der Wert des Elements für den Algorithmus besondere Bedeutung hat, werden mit einem rechteckigen Rahmen um das Element dargestellt. Diese Darstellung ist aus dem Skript der Vorlesung übernommen. Der auf dem Element notierte Wert soll durch den Rahmen hervorgehoben werden. Der Name der Referenz wird direkt am Rahmen, oder in der Legende notiert.

(2) Referenzen, die wegen der Position des Elements in der Liste gewählt sind, werden als senkrechter Pfeil, der auf das Element zeigt, dargestellt. Der Name der Referenz wird an diesem Pfeil notiert. So soll die Position des Elements, auf das die Referenz verweist, leicht erkennbar sein.

Die Unterscheidung ist nicht immer eindeutig und wird anhand dieser Kriterien nach Ermessen getroffen. Falls eine Referenz klar in beiden Kategorien liegt wird die Darstellung als Rahmen entsprechend (1) bevorzugt. Abbildung 3.1a zeigt die Darstellung einer Liste und die von *Selectionsort* verwendeten Referenzen i, j und `minimum` auf Elemente der Liste. Ändert sich eine Referenz, wird die entsprechende Darstellung mit einer kurzen Animation in gerader Linie zum neuen Element verschoben. Die Funktion `Merge` von *Mergesort* verwendet sehr

viele Referenzen, zur Übersichtlichkeit werden daher gerade relevante Referenzen durch schwarze Schriftfarbe gegenüber anderen mit grauer Schriftfarbe hervorgehoben. Abbildung 3.1b zeigt das Kopieren des Ergebnisses in die zu sortierende Liste (siehe Algorithmus 3 Zeile 16). Die Referenz h wird an dieser Stelle der Funktion zum Kopieren verwendet und hat daher besondere Bedeutung.

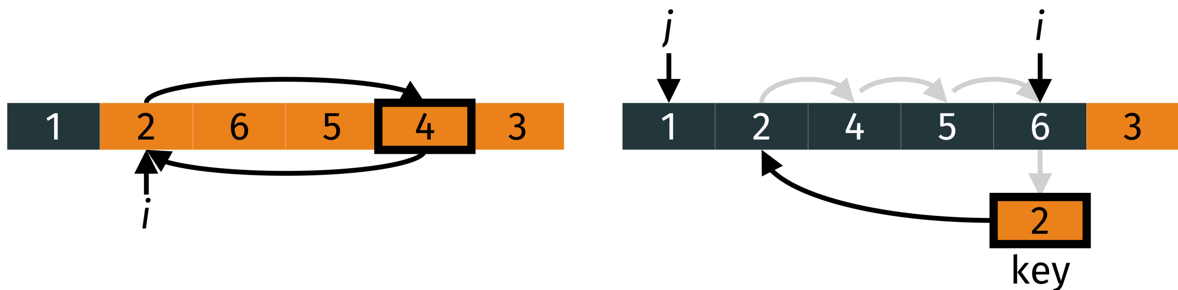


(a) Referenzen i, j und `minimum` (b) Hervorhebung der Referenz h bei *Mergesort*

Abbildung 3.1: Darstellung von Referenzen auf Elemente der Liste

Lokale Variablen, Funktionsargumente und Schleifenzähler der Algorithmen werden nur so lange angezeigt, wie sich der Algorithmus in einem Bereich befindet in dem sie gültig sind. Sobald der Algorithmus den entsprechenden Bereich, die Funktion oder Schleife verlässt, werden zugehörige Elemente aus der Darstellung entfernt.

Das Verschieben, Vertauschen und Kopieren von Elementen wird mit Pfeilen zwischen ihnen dargestellt. Pfeile, die innerhalb einer Liste nach rechts zeigen, werden oberhalb, Pfeile die innerhalb einer Liste nach links zeigen unterhalb der Liste gezeichnet. Abbildung 3.2a zeigt das Vertauschen von Elementen einer Liste mit den entsprechenden Pfeilen. Werden Elemente



(a) Vertauschen von Elementen einer Liste (b) Einfügen des Elements `key` in die Liste

Abbildung 3.2: Pfeile zwischen Elementen von Listen

zwischen verschiedenen Listen verschoben oder zugewiesen, sind die Pfeile zwischen den Listen, unabhängig von ihrer Richtung. Die Elemente werden in der Darstellung mit einer kurzen Animation entlang ihrer zugehörigen Pfeile verschoben. In der Regel werden bei jedem Schritt des Algorithmus die Pfeile des letzten Schritts aus der Darstellung entfernt. Bei *Insertionsort* bleiben auch alte Pfeile in der Darstellung vorhanden, bis das Element `key`

am Ende der äußeren Schleife in die Liste eingefügt wird (siehe Algorithmus 2 Zeile 5), um einen Überblick über die zum Einfügen nötigen einzelnen Kopieroperationen zu ermöglichen. Abbildung 3.2b zeigt das Einfügen des Elements `key` in die Liste. Pfeile aus übersprungenen Schritten bleiben in der Darstellung, damit nachvollzogen werden kann was sich während dieser Schritte am Zustand der Liste verändert hat. Pfeile aus älteren Schritten werden grau, Pfeile des aktuellen Schritts schwarz gezeichnet.

4 Software

Die im Rahmen dieser Arbeit entwickelte Software stellt die Sortierverfahren *Selectionsort*, *Insertionsort*, *Mergesort* und *Quicksort* schrittweise an selbst wählbaren Beispielen dar. Abbildung 4.1 zeigt die erzeugte graphische Benutzeroberfläche. Sie ist horizontal in zwei Bereiche geteilt, im oberen Bereich wird der Zustand der Liste und der Variablen des Sortierverfahrens dargestellt, während im unteren Bereich der Pseudocode des Algorithmus angezeigt und der gerade ausgeführte Schritt darin hervorgehoben wird. Falls das aktive Sortierverfahren selbst Funktionsaufrufe verwendet, wird links neben dem Pseudocode ein Aufrufstapel der Funktionsinkarnationen angezeigt. Über jeweils zwei wählbare Tasten lässt sich der nächste Schritt

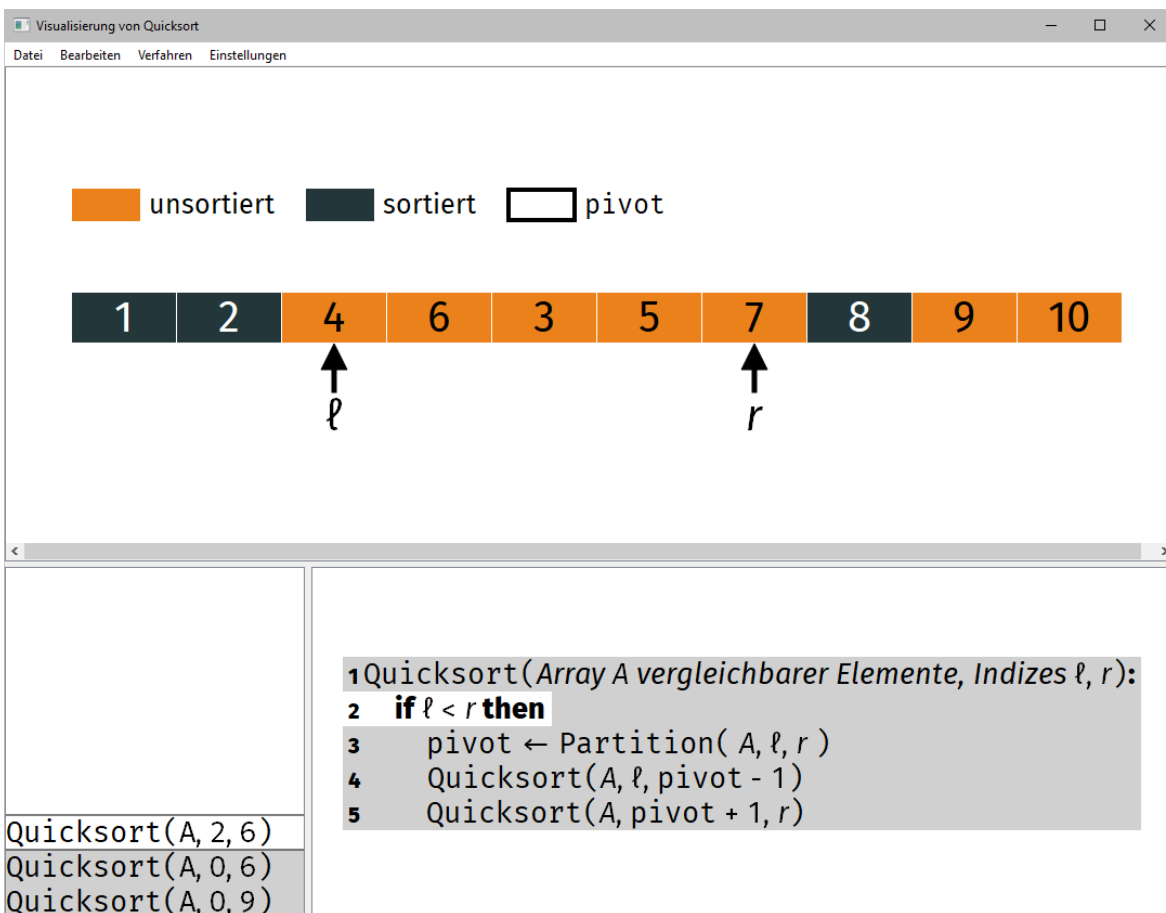


Abbildung 4.1: Graphische Benutzeroberfläche der Software

des Algorithmus ausführen, und zum letzten Zustand zurückspringen. Wird eine der Tasten zum Ausführen des nächsten Schritts für eine vom Benutzer wählbare Zeitspanne gedrückt

gehalten, werden weitere Schritte des Algorithmus ausgeführt, bis das Ende einer Schleife, oder der Beginn der *Merge*- oder *Partition*-Funktion erreicht ist. Mit zwei vom Benutzer wählbaren Tasten lässt sich die Benutzeroberfläche zwischen Fenster- und Vollbildmodus umschalten. Die Elemente der Liste lassen sich über zwei vom Benutzer wählbare Tasten in einer zufälligen Reihenfolge anordnen. Am oberen Rand der Benutzeroberfläche befinden sich vier Menüs mit weiteren Funktionen:

(1) Das Menü „Datei“ erlaubt es, den Zustand des Algorithmus und der Liste als Datei zu speichern und zu laden, den Zustand der Liste als Datei zu speichern und zu laden, und die Darstellung der Liste und der Variablen des Algorithmus als Bilddatei zu speichern.

(2) Im Menü „Bearbeiten“ kann ein Fenster zum Bearbeiten der zu sortierenden Liste geöffnet werden.

(3) „Verfahren“ öffnet eine Liste der implementierten Sortierverfahren, und ermöglicht es zu einem anderen Verfahren zu wechseln.

(4) Über „Einstellungen“ lässt sich ein Fenster zum Bearbeiten einiger Einstellungen des Programms öffnen.

Das Fenster zum Bearbeiten der Liste erlaubt die Bearbeitung der Liste als Text durch Komma getrennter Elemente der Liste. Es enthält Funktionen um die Elemente der Liste pseudozufällig anzuordnen, eine Zeichenkette in einzelne Zeichen zu zerlegen, eine Liste pseudozufälliger Elemente zu generieren, und die Liste als Datei zu speichern oder zu laden.

Im Fenster zum Bearbeiten der Einstellungen kann der Benutzer einige Eigenschaften der Darstellung und Bedienung der Software anpassen. Damit die Software farblich nicht nur zum Skript von „Datenstrukturen und Algorithmen“ passt, sondern auch zu Lehrmaterialien anderer Lehreinrichtungen, lassen sich die Farben der Listenelemente in der Darstellung vom Benutzer wählen. Der Benutzer kann festlegen, ob in der Darstellung eine Legende angezeigt wird, ob die Größe der Darstellung und des Pseudocodes vom Programm gewählt wird und ob die Benutzeroberfläche im Fenster- oder Vollbildmodus startet. Die Tasten für das Ausführen des nächsten Schritts des Algorithmus, das Wiederherstellen des letzten Zustands, den Wechsel zwischen Fenster- und Vollbildmodus und zum zufälligen Anordnen der Listenelemente können in diesem Fenster gewählt werden. Außerdem kann die Zeitspanne, über die eine der Tasten zum Ausführen des nächsten Schritts gedrückt gehalten werden muss, um weitere Schritte zu überspringen, gewählt werden.

4.1 Wahl der Bibliothek für die graphische Benutzeroberfläche

Bei der Wahl einer Bibliothek zur Erzeugung der graphischen Benutzeroberfläche wurden *tkinter*, *PyGObject*, *PyQt6* und *PySide6* betrachtet. Alle diese Bibliotheken laufen auf *Windows*, *macOS* und *Linux*.

> *tkinter* ist ein *Python-Interface* für *Tcl/Tk* [Fou24b] und standardmäßig in Python enthalten. Zusätzlich zur Dokumentation von *tkinter* sind online viele Tutorials verfügbar. *tkinter* unterstützt keine Kantenglättung oder Transparenz für die Darstellung. Kantenglättung ist zwar nicht zwingend erforderlich, aber da für die Darstellung elliptische Pfeile gezeichnet werden sollen wäre die Abwesenheit von Kantenglättung leicht sichtbar und für Betrachter auffällig.

> *PyGObject* sind *Python-Bindings* für *GTK 4* [Tea]. Für *GTK 4* sind online nur wenige Tutorials verfügbar, auf der Webseite wird unter anderem auf *GTK 3 Python*-Tutorials verwiesen. Die Dokumentation von *GTK 4* verwendet in Beispielen *C*. Es erscheint wahrscheinlich, dass Einarbeitung und Fehlersuche bei Verwendung von *PyGObject* erhebliche Zeit in Anspruch nehmen könnten, da online nur sehr wenige Inhalte zu dieser Bibliothek verfügbar sind. *PyGObject* ist unter der LGPL-Linzenz verfügbar[Tea].

> *PyQt6* und *PySide6* sind beide *Bindings* für das *Qt 6 Framework*[Lim][Ltd24], daher ähneln sie sich sehr. Die Dokumentation von *PySide6* enthält für viele Klassen kurze Beispiele in *Python*. Zusätzlich zur jeweiligen Dokumentation sind online einige Tutorials verfügbar. Auch Tutorials für *Qt* in anderen Sprachen können anhand der Beispiele aus der Dokumentation relativ leicht für *Python* angepasst werden. *PyQt6* wird von *Riverbank Computing* entwickelt und ist unter der GPL-Lizenz verfügbar[Lim]. *PySide6* wird wie *Qt* von *The Qt Company* entwickelt und ist unter der LGPL-Lizenz verfügbar[Ltd24].

PySide6 wurde für die Entwicklung der Software gewählt, weil es Kantenglättung und Transparenz unterstützt, durch Dokumentation in Python und genug verfügbare Tutorials eine relativ schnelle Einarbeitung erlaubt und unter einer freieren Lizenz als *PyQt6* verfügbar ist.

4.2 Modularität

Dieser Abschnitt beschreibt die Klassen der entwickelten Software und einige ihrer Attribute und Methoden. Zunächst soll ein Überblick ermöglicht werden, Details folgen in den Unterabschnitten. Alle Klassen und Module außer `Quicksort` und `QuicksortPseudocode`, deren Namen mit „Q“ beginnen, werden von *PySide6* bereitgestellt. Die Darstellung verwendet

`QGraphicsScene` und `QGraphicsItem`. Die Module `copy`, `csv`, `json`, `math`, `random`, `re` und `time` sind Standardmodule von *Python*. Alle anderen Klassen und Module wurden im Rahmen dieser Arbeit implementiert.

Alle Listenelemente und Referenzen auf Listenelemente speichern ihren eigenen Wert, Objekte von `QGraphicsItem` und Animationen der Darstellung gemeinsam in einer Klasse. Die Algorithmen können so durch Methodenaufrufe der Objekte dieser Klasse ihren Wert und die zugehörige Darstellung gleichzeitig verändern. Sortierverfahrenspezifische Klassen sind für jedes Verfahren in einem eigenen Modul zusammengefasst. Für jede Funktion eines Sortierverfahrens wird eine eigene Klasse implementiert, Objekte dieser Klasse führen bei Aufruf der `doStep`-Methode den nächsten Schritt der Funktionsinkarnation aus. Jede Inkarnation der Funktion im Ablauf des Algorithmus entspricht einem Objekt der jeweiligen Klasse auf dem Stapel `Callstack`. Der Pseudocode der Algorithmen ist für jede Funktion des Algorithmus in einer eigenen Klasse implementiert. Allgemeine Eigenschaften und Methoden der Pseudocodedarstellung sind in der Klasse `Pseudocode` zusammengefasst. Von dieser erben die einzelnen Pseudocodeklassen der Funktionen der Algorithmen. Sie enhalten selbst nur Methoden um den Text festzulegen und Schritte hervorzuheben. Zum Speichern und Laden von Zuständen der Sortierverfahren haben einige Klassen eine `getState`-Methode und eine statische `fromState`-Methode. Verwendete Zustände sind *Python*-Listen oder Tupel von Werten, die kombiniert werden können. Der Zustand einer Funktionsinkarnation eines Algorithmus enthält unter anderem die Zustände der in dieser Inkarnation gepeicherten Referenzen auf Elemente der Liste.

4.2.1 Hauptklasse

Die Hauptklasse der Software ist die Klasse `MainWindow`, sie erbt von `QMainWindow` und erzeugt das Fenster der graphischen Benutzeroberfläche. Die Eingabe des Benutzers wird, abgesehen von den Fenstern für das Bearbeiten der Liste und der Einstellungen der Software, in dieser Klasse verarbeitet. Die Klasse enthält Methoden um den Programmablauf zu steuern, z. B. das Sortierverfahren zu wechseln.

Im Konstruktor werden die verwendeten Schriftarten aus den Dateien im Ordner *fonts* geladen, die Einstellungen der Software geladen und die graphische Benutzeroberfläche initialisiert. Die Menüs und Menüpunkte der Software werden vom Konstruktor erzeugt. Ihnen werden für ihre jeweiligen Aktionen entsprechende Methoden der Software übergeben, die Bibliothek ist dann selbst in der Lage diese bei Eingabe des Benutzers aufzurufen.

Die Eingabe des Benutzers durch Tasten wird in den Methoden `keyPressEvent`, `keyReleaseEvent` und `focusOutEvent` behandelt. Sie überschreiben Methoden der Superklasse `QMainWindow`, und werden durch die Bibliothek *PySide6* bei Eingaben des Benutzers aufgerufen. In `keyPressEvent` wird die gedrückte Taste mit den vom Benutzer gewählten Tasten für die verschiedenen Aktionen verglichen, und die entsprechende Aktion durch Aufruf einer Me-

thode ausgeführt. Zum Überspringen mehrerer Schritte des Verfahrens muss eine der Tasten für „Schritt vor“ gedrückt gehalten werden. Beim Drücken der Taste wird in der Methode `keyPressEvent` ein Objekt von `QTimeLine` erzeugt und in `MainWindow.inputtimerforward` gespeichert, das nach einer festgelegten Dauer das Überspringen von Schritten initiiert. Sobald der Benutzer die Taste wieder loslässt, wird die Methode `keyReleaseEvent` durch die Bibliothek aufgerufen. Diese stoppt, falls die losgelassene Taste mit den gewählten Tasten für „Schritt vor“ übereinstimmt, die `QTimeLine` in `MainWindow.inputtimerforward` und verhindert damit das spätere Überspringen von Schritten. Falls der Benutzer zu einem anderen Fenster wechselt, muss `MainWindow.inputtimerforward` ebenfalls gestoppt werden, weil dieses Fenster Eingaben des Benutzers dann nicht mehr erhält. Die Methode `focusOutEvent` wird in diesem Fall von der Bibliothek aufgerufen. Da Objekte vom Typ `QGraphicsView` für die Darstellung der Liste, des Pseudocodes und des Aufrufstapels einige Eingaben selbst bearbeiten würden, anstatt sie an die Hauptklasse weiterzureichen, wird eine Klasse `GraphicsView` verwendet, die von `QGraphicsView` erbt und die Methoden `keyPressEvent`, `keyReleaseEvent` und `focusOutEvent` überschreibt.

Veränderungen der Größe des Fensters erfordern eine Anpassung der Größe der Darstellungen der Liste, des Pseudocodes und des Aufrufstapels, die in der Methode `reDraw` implementiert ist und im Abschnitt 4.3 Größenbeschränkungen genauer beschrieben wird.

Das Speichern und Laden von Zuständen (siehe Unterabschnitt 4.2.7) der Sortierverfahren und die Zurück-Funktion (siehe Unterabschnitt 4.2.8) werden durch einen Aufruf von Methoden dieser Klasse initiiert, aber später in eigenen Unterabschnitten beschrieben.

4.2.2 Darstellung der Liste

Abbildung 4.2 zeigt ein Klassendiagramm einiger Klassen der Listendarstellung, das beim Lesen der folgenden Unterabschnitte zur Orientierung dienen soll. Es ist nicht vollständig, sondern zur Übersichtlichkeit vereinfacht. Einige der dargestellten Klassen haben weitere Attribute und Methoden, die weniger relevant sind da sie z. B. nur von der eigenen Klasse aufgerufen werden. Eine Beschreibung aller Attribute und Methoden befindet sich in den *Docstrings* der Klassen im Quellcode. Im Klassendiagramm wird zwischen *Python*-Listen „`list`“ und der Klasse „`List`“ unterschieden. Die Klasse `Merge` ist als Beispiel für die Klassen der Sortierverfahren im Diagramm eingezeichnet, sie implementiert die Funktion `Merge` von *Mergesort*.

Liste

Listen sind in den Klassen `List` und `ListElement` implementiert. Objekte von `List` enthalten ihre Listenelemente als `ListElement` in einer *Python*-Liste `List.arr`. Diese Elemente speichern ihren eigenen Wert, und erzeugen Objekte von `QGraphicsItem` zur Darstellung eines Elements der Liste. Sie werden von ihrer Liste erstellt und sollten nur durch Aufruf

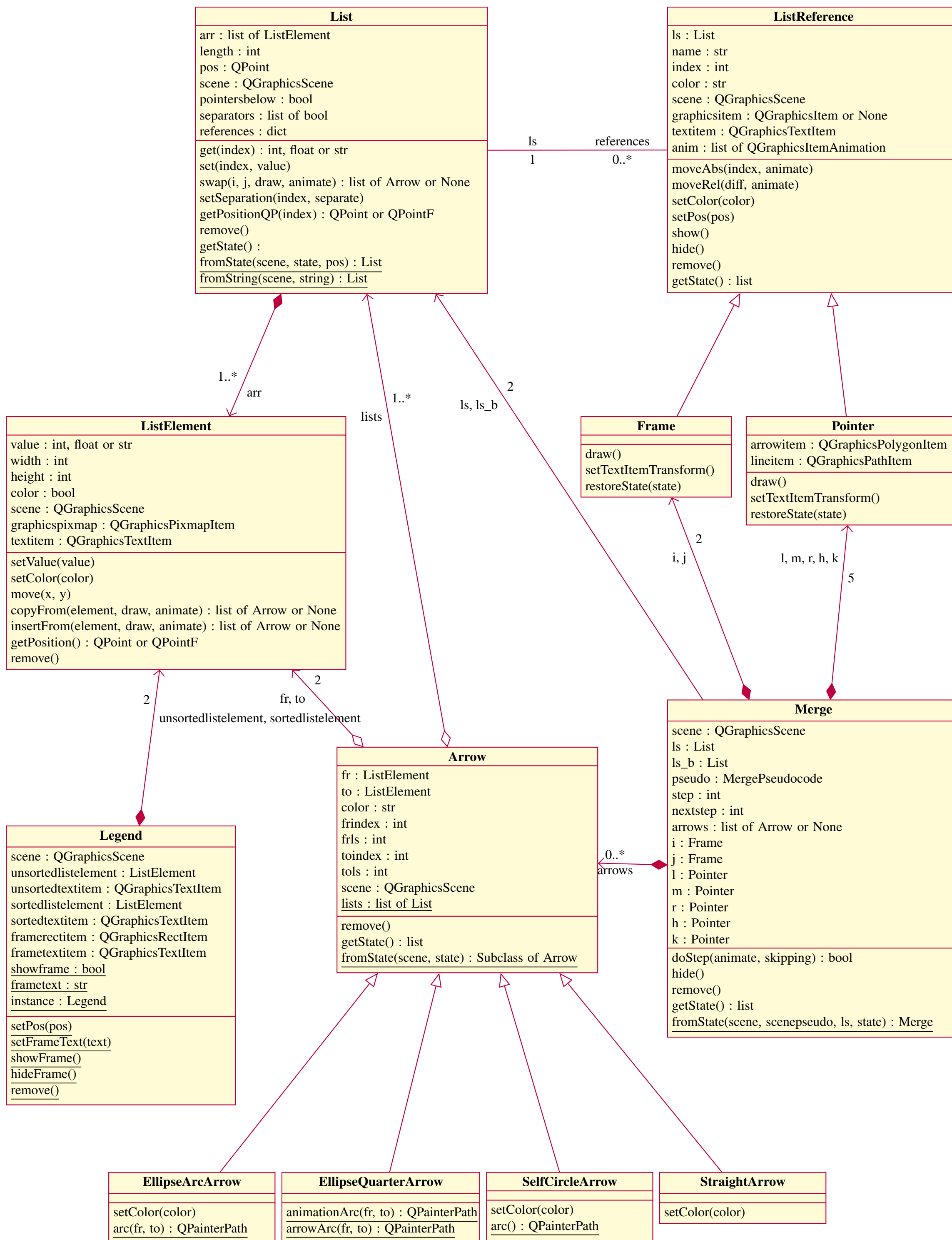


Abbildung 4.2: Klassendiagramm einiger Klassen der Listendarstellung

von Methoden von `List` und `ListElement` bearbeitet werden. `List` implementiert selbst keine Darstellung, koordiniert aber die Darstellungen der Listenelemente durch Vorgabe ihrer Position in der Szene und der Breite der einzelnen Elemente. Änderungen an der Liste können durch die Sortierverfahren über Methoden von `List`, falls mehrere Elemente verändert werden, und `ListElement`, falls nur ein Element verändert wird, veranlasst werden. Diese Methoden erzeugen, falls erwünscht, auch entsprechende Animationen und Pfeile in der Darstellung. Beispielsweise lassen sich Elemente der Liste mit der Methode `List.swap` vertauschen, als Argumente benötigt sie die Indizes der zu vertauschenden Elemente und Wahrheitswerte für das Erzeugen von Pfeilen und Animationen. Der Aufruf von `swap(0, 5, True, True)` würde das erste und sechste Element der Liste vertauschen, Pfeile zwischen den Elementen erzeugen und die Elemente in der Darstellung mit einer Animation entlang dieser Pfeile bewegen. Die erzeugten Pfeile werden in einer *Python*-Liste als Rückgabewert der Methode zurückgegeben.

Verweise auf Elemente der Liste

Alle implementierten Sortierverfahren arbeiten mit Verweisen auf Elemente der Liste. Sie sind in der Klasse `ListReference` implementiert, ihre Anzeige in der Darstellung wird durch die Subklassen `Frame` und `Pointer` festgelegt, die den im Kapitel 3 Darstellung beschriebenen Varianten (1) und (2) entsprechen. Objekte dieser Klassen werden von den Sortierverfahren selbst erstellt und verändert. Sie enthalten einen Namen für die Anzeige, eine Farbe für den Namen in der Darstellung, die Liste in der sich das Listenelement auf das sie verweisen befindet und den Index des Elements auf das sie verweisen in seiner Liste. Um mit einer bestehenden Referenz auf ein anderes Element der selben Liste zu verweisen, werden die Methoden `ListReference.moveAbs` und `ListReference.moveRel` verwendet. Sie benötigen als Argumente den Index des neuen Elements oder die Differenz der Indizes des neuen und vorigen Elements, und einen Booleschen Wahrheitswert dafür ob die Darstellung der Referenz in einer Animation zum neuen Listenelement verschoben werden soll.

Ob die Namen einer Referenz über oder unter der Liste angezeigt werden, wird in dem Attribut `List.pointersbelow` vom Typ `bool` durch die Sortierverfahren festgelegt. Ist dieses `True`, werden die Namen und Pfeile von `Pointer` unterhalb und die Namen von `Frame` oberhalb der Liste angezeigt, andernfalls jeweils auf der anderen Seite. Damit die Namen von Referenzen auch vor Pfeilen lesbar bleiben, werden sie mit einem weißen Hintergrund hinterlegt, der gegebenenfalls einen Teil des Pfeils verdeckt. Damit die Namen mehrerer Referenzen des gleichen Typs, die auf das gleiche Element verweisen, alle lesbar bleiben, werden sie in einer durch Komma getrennten Liste ihrer Namen dargestellt. `List` enthält dazu im Attribut `List.references` ein *Python-Dictionary*, das für jeden Typ von Referenz ein *Python-Dictionary* mit einer *Python*-Liste von Referenzen für jedes Element der Liste enthält. In diese Listen tragen sich Objekte von `ListReference` automatisch ein und aus, wenn sie

z. B. auf ein neues Element verweisen, oder sichtbar oder verborgen werden. Die Namen aller Elemente dieser Listen werden dann anstelle des Namens der ersten Referenz in der Liste angezeigt, und die Namen aller anderen Referenzen in der Liste in der Darstellung verborgen.

Pfeile

Pfeile zwischen Elementen von Listen sind im Modul `Arrow` implementiert. Pfeile werden von den Methoden von `List` und `ListElement` erzeugt, die eine durch Pfeile darstellbare Aktion auf Elementen der Liste ausführen. Verschiedene Arten von Pfeilen sind in eigenen Klassen implementiert. Das Kopieren von Elementen zwischen verschiedenen Listen wird durch `StraightArrow` dargestellt. Beim Vertauschen von Elementen einer Liste, oder Kopieren eines Elements der selben Liste wird `EllipseArcArrow` verwendet. Vertauschen eines Elements mit sich selbst wird von `SelfCircleArrow` dargestellt. Für Insertionsort wird beim Kopieren von `key` zurück in die zu sortierende Liste ein `EllipseQuarterArrow` verwendet. Den Konstruktoren der Pfeilklassen werden die betroffenen Listenelemente übergeben, anhand deren Position wird der Pfeil in der Darstellung gezeichnet. `EllipseArcArrow`, `EllipseQuarterArrow` und `SelfCircleArrow` zeichnen ihren Pfeil entlang eines Pfades vom Typ `QPainterPath`, der jeweils von statischen Methoden der Klassen erzeugt wird. Für die Animation von Listenelementen entlang eines Pfeils werden ebenfalls Objekte von `QPainterPath` verwendet. Für `EllipseQuarterArrow` müssen zum Zeichnen und Animieren etwas verschiedene Pfade verwendet werden, da der gezeichnete Pfeil nicht an den selben Stellen der Elemente beginnt und endet. Die Methoden zum Erzeugen der Pfade für `EllipseQuarterArrow` sind `arrowArc` zum Zeichnen und `animationArc` für Animationen. `EllipseArcArrow` und `SelfCircleArrow` haben stattdessen jeweils nur eine Methode `arc`.

Die Klassen der verschiedenen Pfeilarten erben von `Arrow` die Methoden `getState` und `fromState`, um Pfeile in der Darstellung als Teil des Sortierzustands zu speichern und laden. Dazu werden die Indizes der Listenelemente, zwischen denen der Pfeil gezeichnet ist, in ihrer Liste verwendet. Um die jeweilige Liste der Elemente wieder zuordnen zu können, wird der Index der Listen in einer *Python*-Liste aller vom Sortierverfahren verwendeten Listen für beide Elemente verwendet. Diese Liste aller verwendeten Listen ist in der statischen Variable `Arrow.lists` gespeichert, und wird vom Sortierverfahen festgelegt, falls nicht nur die zu sortierende Liste verwendet wird. Die Reihenfolge der Listen in `Arrow.lists` muss von den Sortierverfahren beachtet werden. Sie sollte unabhängig davon, ob der Zustand des Verfahrens aus der Ausführung resultiert oder gepeichert und wiederhergestellt wurde, gleich sein.

Legende

Die Legende der Darstellung ist in der Klasse `Legend` implementiert. Objekte der Klasse werden von der Hauptklasse erstellt. Sie erzeugen selbst eigene Objekte von `QGraphicsItem`

zur Darstellung in der Szene. Es gibt immer nur ein Objekt von `Legend`, das gleichzeitig dargestellt wird. Es ist in der statischen Variable `Legend.instance` gespeichert. Wenn weitere Objekte erstellt werden, entfernt deren Konstruktor das ältere Objekt aus der Darstellung. Für die Darstellung von bereits sortierten und unsortierten Listenelementen in der Legende werden zwei Objekte von `ListElement` verwendet. Durch Aufruf der Methoden `setPos`, `setFrameText`, `showFrame` und `hideFrame` der Klasse `Legend` legen die Sortierverfahren die Position und den Inhalt der Legende fest. Die Legende wird so erstellt, dass sie etwas kleiner ist als die Darstellung der Liste, sie nimmt etwa zwei Drittel einer Zeile der Darstellung der Liste ein. Sie wird trotzdem eine ganze Zeile oberhalb der Listendarstellung positioniert, um sie von der Darstellung räumlich zu trennen.

4.2.3 Algorithmen

Wie anfangs erwähnt sind die Sortierverfahren jeweils in einem eigenen Modul implementiert. Jede Funktion der Sortierverfahren ist in einer Klasse der Software implementiert. Jede Funktionsinkarnation des Verfahrens entspricht einem Objekt der Klasse. Die Funktionsargumente und lokalen Variablen der Inkarnation sind Attribute des Objekts. Durch Aufruf der `doStep`-Methode des Objekts wird der nächste Schritt der Funktion ausgeführt. Die Methode hat zwei optionale Argumente `animate` und `skipping` vom Typ `bool`, die angeben, ob für den ausgeführten Schritt Animationen in der Darstellung erzeugt werden sollen, und ob der Schritt beim Überspringen von Schritten oder strikt schrittweise ausgeführt wird. Die implementierten Verfahren färben zunächst alle Pfeile aus vorherigen Schritten grau, falls `skipping` `True` ist, andernfalls entfernen sie alle Pfeile aus der Darstellung. Die einzelnen Schritte werden von den implementierten Verfahren durch eine Zahl vom Typ `int` unterschieden, der aktuelle und nächste Schritt sind in den Attributen `step` und `nextstep` der Funktionen entsprechenden Klassen gespeichert. Die Methode `doStep` speichert `nextstep` in `step`. Im Pseudocode wird der entsprechende Schritt durch Aufruf von `Pseudocode.highlight` hervorgehoben. Die verschiedenen Schritte sind in einem *Match-Case-Statement* implementiert, in dem der Schritt `step` anschließend ausgeführt wird. Dabei wird unter anderem der nächste Schritt `nextstep` festgelegt. Falls nötig werden darin auch weitere Schlüsselwörter im Pseudocode hervorgehoben, beispielsweise *then* oder *else*, je nachdem, ob die Bedingung eines *If-Statements* wahr oder falsch ist. Der Rückgabewert von `doStep` gibt an, ob das Überspringen von Schritten weiter fortgeführt oder unterbrochen werden soll. `False` wird am Ende einer Schleife und dem Beginn der inneren Funktionen (`Merge` bei *Mergesort* und `Partition` bei *Quicksort*) zurückgegeben und signalisiert das Ende des Überspringens.

Fallunterscheidungen der Sortierverfahren sind einfach durch *If-Statements* in *Python* implementiert, die den nächsten Schritt `nextstep` je nach Ergebnis der Bedingung unterschiedlich festlegen. Schleifen werden durch zyklische Schrittfolgen erzeugt. Der Beginn einer Schleife kann von den späteren Wiederholungen beispielsweise durch die Sichtbarkeit

des Schleifenzählers erkannt werden. Wird eine Schleife nicht erneut wiederholt, gibt `doStep` `False` zurück, um das Überspringen von Schritten zu beenden.

Die Klasse `Callstack` implementiert einen Aufrufstapel für Funktionsinkarnationen der Sortierverfahren, bei einem Funktionsaufruf der Sortierverfahren erstellt das der aufrufenden Funktionsinkarnation entsprechende Objekt per Konstruktor ein neues, der aufgerufenen Funktionsinkarnation entsprechendes, Objekt und legt dieses mit der Methode `Callstack.push` auf den Stapel. Alle Elemente der Darstellung der aufrufenden Funktion müssen beim Funktionsaufruf mittels ihrer Methode `hide` verborgen werden. Am Ende eines Funktionsaufrufs entfernt sich das der beendeten Funktionsinkarnation entprechende Objekt durch Aufruf von `Callstack.pop` vom Aufrufstapel. Falls ein Rückgabewert übergeben wird, speichert das der endenden Funktionsinkarnation entsprechende Objekt ihn in der statischen Variable `Algorithm.returnvalue`. Das der aufrufenden Funktionsinkarnation entsprechende Objekt kann diesen Wert anschließend lesen. Nach dem Ende eines Funktionsaufrufs zeigt das der aufrufenden Funktionsinkarnation entsprechende Objekt alle zuvor verborgenen Elemente der Darstellung durch Aufruf ihrer `show`-Methoden wieder an. Um im selben Schritt zwischen Beginn und Ende des Funktionsaufrufs zu unterscheiden wird die Sichtbarkeit der Elemente der Darstellung verwendet. Sind sie sichtbar, wird die Funktion aufgerufen, sind sie nicht sichtbar wird der Rückgabewert aus `Algorithm.returnvalue` gelesen und der nächste Schritt festgelegt.

Der Pseudocode ist in eigenen Klassen implementiert, eine Superklasse `Pseudocode` implementiert Pseudocode allgemein. Für jede Funktion des Sortierverfahrens muss eine Subklasse den Text und das Hervorheben der einzelnen Schritte und Schlüsselwörter durch Überschreiben der Methoden `assembleText` und `addHighlight` festlegen. `Pseudocode` stellt in statischen Variablen Schriftarten für den Text bereit. Objekte der Pseudocodeklassen werden im Konstruktor der den Funktionsinkarnationen entsprechenden Objekten erzeugt.

4.2.4 Darstellung des Aufrufstapels

Die Darstellung des Aufrufstapels ist in der Klasse `CallstackDisplay` implementiert. Ein Objekt der Klasse wird von der Hauptklasse erstellt. Ähnlich wie bei der Legende in Unterabschnitt 4.2.2 wird nur eine Instanz des Aufrufstapels gleichzeitig angezeigt. Falls weitere Objekte von `CallstackDisplay` erstellt werden, entfernt deren Konstruktor ältere Objekte aus der Darstellung. Die Darstellung wird von der Hauptklasse nur bei Sortierverfahren angezeigt, die mehrere Funktionsaufrufe verwenden, sonst ist sie verborgen. `Callstack` ruft bei Veränderungen am Aufrufstapel die entsprechenden Methoden `CallstackDisplay.push` und `CallstackDisplay.pop` auf. Die Funktionsinkarnationen werden jeweils in einer Zeile übereinander dargestellt, der Text wird durch die `toString`-Methoden der Klassen der Sortierverfahren festgelegt. Ist der Stapel höher als der verfügbare Platz für die Darstellung, wird nur der obere Teil des Stapels angezeigt. Bei Änderungen am Stapel wird die Darstel-

lung dann durch eine Animation entsprechend nach oben oder unten verschoben. Wenn das laufende Sortierverfahren die Liste fertig sortiert hat und beendet wird, ruft die Hauptklasse `CallstackDisplay.onAlgorithmComplete` auf. Diese Methode ersetzt den dargestellten Text der Funktionsinkarnation auf dem Stapel durch den Text "fertig", um dem Benutzer das Ende der Ausführung zu signalisieren.

4.2.5 Einstellungen

Einstellungen der Software werden in statischen Variablen der Klasse `Settings` gespeichert. Um sie über die aktuelle Programmausführung hinweg verwenden zu können, werden sie bei Veränderungen durch den Benutzer als Werte eines *Python dictionaries* zusammengefasst und mit dem Modul `json` in eine Zeichenkette übersetzt als Textdatei gespeichert. Beim Start des Programms werden die Einstellungen zunächst mit Standardwerten initialisiert, und dann falls verfügbar mit Werten aus dieser Datei überschrieben. Einige nur für die aktuelle Programmausführung benötigte globale Variablen sind ebenfalls als statische Variablen der Klasse `Settings` implementiert, diese werden nicht gespeichert, und mit `None` inititalisiert. Die Werte werden dort von anderen Klassen gespeichert.

Ein Fenster zum Bearbeiten der Einstellungen lässt sich vom Benutzer über den Menüpunkt „Einstellungen...“ des Menüs „Einstellungen“ öffnen. Die Oberfläche des Fensters wurde mit dem von *PySide6* bereitgestellten Programm *pyside6-designer* erzeugt, die von diesem Programm gespeicherte Datei `Ui_SettingsDialog.ui` wurde durch das Skript `pyside6-uic` in das *Python*-Modul `Ui_SettingsDialog` übersetzt. Die Funktion der verschiedenen Elemente der Benutzeroberfläche sind in der Klasse `SettingsDialog` implementiert, sie erbt von `QWidget` und `Ui_settingsdialog`. Die Hauptklasse erzeugt in ihrem Konstruktor ein Objekt von `SettingsDialog`, das entsprechende Fenster der Benutzeroberfläche kann dann beliebig oft angezeigt und wieder verborgen werden. Änderungen an den Einstellungen durch das Programm müssen z. B. durch Aufruf der Methode `initializeValues` in den Elementen der Benutzeroberfläche aktualisiert werden.

Die vom Benutzer wählbaren Einstellungen der Darstellung sind die Farben der Schrift, des Hintergrundes und des Randes der Listenelemente in der Darstellung jeweils für unsortierte und bereits sortierte Elemente, ob eine Legende der Darstellung angezeigt wird, ob die Größe der Darstellung vom Benutzer oder durch das Programm selbst gewählt wird und ob das Programm im Vollbildmodus startet. Falls der Benutzer die Größe der Darstellung festlegt, wählt er außerdem die Schriftgröße der Darstellung, andernfalls wird diese vom Programm anhand der Größe des Fensters gewählt. Für die Bedienung kann der Benutzer jeweils zwei Tastenkombinationen für die Aktionen „Schritt vor“, „Schritt zurück“, zum Wechsel zwischen Vollbild- und Fenstermodus und zum zufälligen Anordnen der Elemente der Liste wählen, und die Zeitspanne, ab der eine gedrückte Taste als gedrückt gehalten gilt, festlegen. Änderungen des Benutzers an den Einstellungen werden von `SettingsDialog` sofort umgesetzt.

4.2.6 Bearbeiten der Liste durch den Benutzer

Das Bearbeiten der Liste durch den Benutzer ist in der Klasse `EditList` implementiert. Analog zu `SettingsDialog` erbt diese von `QWidget` und `Ui_editlist`. `Ui_editlist` enthält die mit *pyside6-designer* erstellte Benutzeroberfläche, die vom Skript `pyside6-uic` in das *Python*-Modul `Ui_EditList` übersetzt wurde. Ein Objekt von `EditList` wird vom Konstruktor der Hauptklasse erstellt, und dann auf Eingabe des Benutzers über den Menüpunkt „Liste bearbeiten...“ im Menü „Bearbeiten“ angezeigt und beim Schließen des Fensters verborgen. Bei jeder Änderung der Liste müssen die Elemente der Benutzeroberfläche z. B. durch Aufruf der Methode `initializeValues` aktualisiert werden.

Änderungen des Benutzers an der Liste werden durch Methoden von `EditList` und `MainWindow` sofort umgesetzt, eine Bestätigung ist nicht erforderlich. Sie können aber durch die „Schritt zurück“-Funktion rückgängig gemacht werden. Das Objekt der Klasse `List` wird nicht verändert sondern durch ein Neues ersetzt. Das laufende Sortierverfahren wird bei Änderungen an der Liste unterbrochen, und vollständig durch neue Objekte ersetzt. Die Ausführung beginnt erneut am Anfang des Verfahrens.

Das Speichern und Laden der Liste als Datei ist in den Methoden `saveList` und `loadList` der Hauptklasse implementiert. Zum Speichern werden die Werte der Listenelemente mit dem Modul `csv` als durch Komma getrennte Zeichenkette in einer Datei gespeichert. Beim Laden einer solchen Datei wird die Liste durch die Methode `List.fromString` aus der Zeichenkette wiederhergestellt, und die zu sortierende Liste durch diese neue Liste ersetzt. Wie bei Änderungen des Benutzers an der Liste beginnt das laufende Sortierverfahren auch beim Laden der Liste wieder von vorn.

4.2.7 Speichern und Laden von Zuständen

Das Speichern und Laden von Zuständen wird von der Hauptklasse initiiert und ist jeweils in zwei Schritte geteilt. Bei Aufruf der `getState`-Methode von `MainWindow` werden alle Werte des aktuellen Sortierzustands aus den Objekten des laufenden Programms in eine Datenstruktur im Hauptspeicher kopiert. Durch Aufruf der `restoreState`-Methode von `MainWindow` lässt sich dieser Zustand in der laufenden Programmausführung wiederherstellen. Zum Speichern dieser Datenstruktur als Datei wird sie mit dem Modul `json` in eine Zeichenkette übersetzt, die als Textdatei gespeichert wird. Beim Laden solcher Dateien kann das Modul die Datenstruktur aus der entsprechenden Zeichenkette wiederherstellen. Nach der Dokumentation [Fou24a] darf die Datenstruktur für diese Übersetzung zu Zeichenketten und zurück nur Objekte der Typen `dict`, `list`, `tuple`, `str`, `int`, `float` und von `int` oder `float` abgeleitete `Enums` und die Werte `True`, `False` und `None` enthalten. Die Zustände der Objekte, die einer Funktionsinkarnation der Sortierverfahren entsprechen, enthalten als erstes Element die Klasse des Objekts, um den Zustand beim Wiederherstellen der richtigen

Klasse zuordnen zu können. Diese lässt sich durch das Modul `json` nicht automatisiert übersetzen, daher wird sie beim Speichern als Datei in der `MainWindow.saveState`-Methode mit einem eindeutigen Namen der Klasse ersetzt, und beim Laden aus einer Datei in der `MainWindow.loadState`-Methode anhand dieses Namens wiederhergestellt.

Der Zustand, den `MainWindow.getState` zurückgibt, ist eine Liste des Zustands der zu sortierenden Liste, des Wertes von `Algorithm.returnvalue` und des Zuständs des Ausführungsstapels `Callstack`. Die `getState`-Methoden verschiedener Klassen sind untereinander verschachtelt, die `getState`-Methode der Hauptklasse ruft die `getState`-Methoden der Liste `List` und des Aufrufstapels `Callstack` auf. `Callstack.getState` ruft die `getState`-Methoden der den Funktionen der Sortierverfahren entsprechenden Klassen auf, diese verwenden die `getState`-Methoden der Klassen `Arrow`, `ListReference` und eventuell `List` wenn sie zusätzliche Listen verwenden.

4.2.8 Zurück-Funktion

Für die in der Aufgabenstellung festgelegte Zurück-Funktion, die schrittweise vergangene Zustände des Sortierverfahrens wiederherstellen soll, wurden zwei Ansätze zur Implementierung betrachtet.

(1) Es wäre möglich, vergangene Zustände im Hauptspeicher zu speichern, um sie später wiederherstellen zu können. Dazu wäre natürlich Speicherplatz im Hauptspeicher erforderlich, und die Anzahl gespeicherter Zustände wäre durch den verfügbaren Speicher beschränkt. Auf diese Weise könnten auch Zustände vor einem später vorgenommenen Wechsel des Sortierverfahrens wiederhergestellt werden, und versehentliche Änderungen der Liste durch den Benutzer wieder verworfen werden. Die Implementierung könnte gemeinsam mit der Implementierung des Speicherns und Ladens von Zuständen der Sortierverfahren als Datei erfolgen, die ohnehin erforderlich ist, und würde so keinen zusätzlichen Aufwand erfordern.

(2) Eine Rückwärts-Implementierung der Sortierverfahren könnte ohne zusätzlichen Speicherplatz im Hauptspeicher auskommen, wäre aber nicht in der Lage Zustände des zuletzt verwendeten Sortierverfahrens nach einem Verfahrenswechsel wiederherzustellen. Außerdem wäre eine solche Implementierung nur für deterministische Algorithmen möglich, und würde erheblichen zusätzlichen Aufwand beim Implementieren weiterer Sortierferfahren bedeuten.

Ansatz (1) wurde für die Implementierung der Software gewählt, weil er dem Benutzer mit dem Wiederherstellen von Zuständen anderer Algorithmen nach einem Wechsel des Sortierverfahrens und dem Rückgängigmachen von Änderungen an der zu sortierenden Liste mehr Möglichkeiten als Ansatz (2) bietet und weniger Aufwand als Ansatz (2) erfordert. Außerdem würde Ansatz (2) für das hinzufügen weiterer Sortierverfahren zur Software

deutlich mehr Aufwand erfordern, und könnte für manche Sortierverfahren, z. B. Quicksort mit zufälligem Pivotelement, nicht ohne weiteres verwendet werden.

Das Wiederherstellen vorheriger Sortierzustände der aktuellen Programmausführung ist in der Klasse `MainWindow` implementiert. Die Variable `MainWindow.history` enthält eine Liste vergangener Zustände und des aktuellen Zustands. Die Methode `MainWindow.addToHistory` fügt `MainWindow.history` einen weiteren Zustand hinzu, und entfernt den ältesten gespeicherten Zustand falls `MainWindow.history` nun mehr als `MainWindow.historylimit` Zustände enthält. Der aktuelle Sortierzustand `MainWindow.getState` wird auf diese Weise nach dem Ausführen eines Schritts, nach dem Überspringen von Schritten, nach dem Laden eines Zustands aus einer Datei, nach dem Laden der zu sortierenden Liste aus einer Datei und nach Änderungen an der zu sortierenden Liste durch den Benutzer gespeichert. Wenn der Benutzer eine der beiden Tasten für „Schritt zurück" drückt, oder im Menü „Bearbeiten" „Schritt zurück" wählt, wird die Methode `MainWindow.stepBack` aufgerufen. Falls `MainWindow.history` mindestens zwei Zustände enthält, den aktuellen und einen vergangenen, entfernt die `MainWindow.stepBack`-Methode den aktuellen Zustand aus `MainWindow.history` und stellt durch einen Aufruf von `MainWindow.restoreState` den letzten Zustand wieder her. Für Beobachter, die die Eingabe des Benutzers nicht selbst sehen können, wird in der Mitte der Listendarstellung ein linksgerichteter Pfeil angezeigt, um darauf aufmerksam zu machen, dass die Änderungen in der Darstellung sich nicht aus einem Schritt vor, sondern einem Schritt zurück ergeben. Dieser Pfeil wird von `MainWindow.graphicsBack` erzeugt und durch `MainWindow.displayAction` angezeigt. Er bleibt in der Darstellung für 750 Millisekunden sichtbar und wird dann automatisch entfernt.

Um eine Obergrenze für die Anzahl gleichzeitig im Hauptspeicher gespeicherter Zustände zu wählen, wurde der von einem Prototypen verwendete Speicher bei verschiedener Anzahl gespeicherter Zustände gemessen. Die Messung ist mit einer 128 Elemente langen Liste ganzer Zahlen des Intervalls $[1000, 9999]$ bei einer Fenstergröße von 640 Pixel Breite und 480 Pixel Höhe auf einem *Windows 10* System erfolgt, der verwendete Hauptspeicher wurde im *Task-Manager* des Betriebssystems abgelesen. Gespeichert wurden schrittweise aufeinander folgende Zustände der Sortierverfahren. Die Liste wurde von den Verfahren Mergesort (nach 8114 Schritten) und Quicksort (nach 4057 Schritten und erneut nach 8115 Schritten) innerhalb des Messvorgangs bereits vollständig sortiert. In diesen Fällen wurde die ursprüngliche, unsortierte Liste wiederhergestellt und erneut sortiert, um weitere Zustände zu speichern.

Abbildung 4.3 zeigt die so gemessene Menge an Hauptspeicher für die einzelnen Sortierverfahren. *Mergesort* und *Quicksort* benötigen deutlich mehr Speicher als *Selectionsort* und *Insertionsort*. Mit dem Ziel, 500 MB an benötigtem Hauptspeicher bei üblichem Gebrauch der Software nicht zu überschreiten, und genug Zustände zu speichern, um in der Regel so oft wie vom Benutzer gewünscht zurück springen zu können, wurde eine obere Schranke der Anzahl im Hauptspeicher gespeicherter Zustände von 8192 Zuständen gewählt. Das bedeutet, dass der Benutzer 8191 vergangene Zustände wiederherstellen kann, und sollte für den vorge-

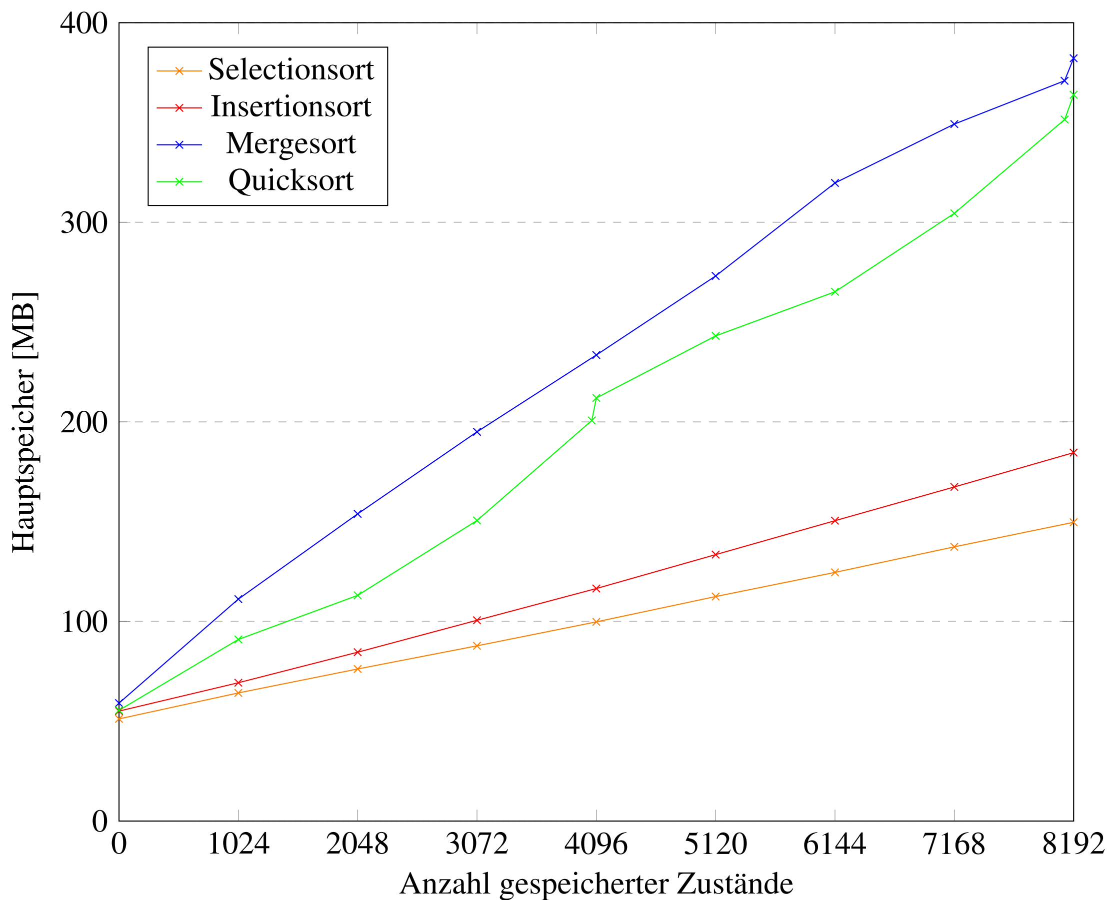


Abbildung 4.3: Verwendeter Haupspeicher in Abhängigkeit der Anzahl im Hauptspeicher gespeicherter Zustände

sehenen didaktischen Gebrauch der Software vollkommen ausreichen. Eine Optimierung des Speicherbedarfs wäre z. B. durch Speichern der Änderungen zwischen Zuständen anstatt des ganzen Zustands möglich, erscheint aber anhand des gemessenen Speicherbedarfs nicht als notwendig.

4.3 Größenbeschränkungen

Um die Größe der Darstellung der Liste, des Pseudocodes und des Aufrufstapels festzulegen, wird primär die für diese Elemente zur Verfügung stehende Höhe betrachtet. Für die Darstellung der Liste werden maximal acht Zeilen Text benötigt, falls die Legende nicht angezeigt wird, und maximal neun Zeilen, falls die Legende angezeigt wird. Für die Darstellung des Pseudocodes werden maximal zehn Zeilen benötigt. Ist die Darstellung der Liste breiter als das Fenster, lässt sie sich mit einer Scrollleiste horizontal verschieben. Falls der Pseudocode breiter als zulässig ist, wird er zur verfügbaren Breite skaliert. Die Schriftgröße wird anhand der verfügbaren Höhe des Fensters und der maximal benötigten Zeilenanzahlen von Listendarstellung und Pseudocode gewählt. Von der Schriftgröße der Darstellung der Liste hängen

die Größen aller Elemente dieser Darstellung ab, auch solche, die keinen Text enthalten, wie beispielsweise Pfeile oder Rahmen.

Die Schriftgröße von Aufrufstapel und Pseudocode hängt über einen Faktor von der Schriftgröße der Darstellung der Liste ab, dieser Faktor wird je nach Seitenverhältnis gewählt. Bei einem Seitenverhältnis kleiner als 4:3 wird der Aufrufstapel nicht angezeigt, da die geringe Breite eine Darstellung von Aufrufstapel und Pseudocode nebeneinander ungünstig macht. Die Schriftgröße des Pseudocodes ist dann die Schriftgröße der Listendarstellung mal 0,9. Damit Pseudocode und Aufrufstapel beim Seitenverhältnis 4:3 nebeneinander passen, werden sie etwas kleiner angezeigt. Der Faktor für die Schriftgröße ist dann 0,67, und steigt von 4:3 bis 16:9 linear auf 0,9 an. Ab einem Seitenverhältnis von 16:9 bleibt der Faktor 0,9.

Die minimale Größe des Fensters ist auf 640 Pixel Breite und 480 Pixel Höhe festgelegt. Bei dieser Größe ist der Pseudocode mit einer Schriftgröße von 10,7pt noch gut lesbar, kleiner sollte er aber nicht sein. Diese Größe erlaubt es auf modernen Bildschirmen mehrere Instanzen der Software nebeneinander zu betrachten, und sorgt für gute Kompatibilität mit älteren Geräten.

Alternativ kann der Benutzer die Größe der Darstellung durch Festlegung der Schriftgröße in den Einstellungen selbst wählen. Die Darstellung der Liste und des Pseudocodes haben dann Scrollleisten, falls sie größer sind als der verfügbare Platz erlaubt. Außerdem lässt sich die horizontale Trennlinie zwischen Listen- und Pseudocodedarstellung dann vom Benutzer verschieben.

5 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde eine Software entwickelt, die die Sortierverfahren *Selectionsort*, *Insertionsort*, *Mergesort* und *Quicksort* schrittweise ausführt und darstellt. Die Software ermöglicht dem Benutzer das Speichern und Laden von Zuständen als Datei, das Speichern und Laden der zu sortierenden Liste als Datei, das schrittweise Wiederherstellen vergangener Zustände, das Überspringen mehrerer Schritte und das Exportieren der Darstellung als Bilddatei. Darüber hinaus bietet die Software Benutzern einige Einstellungsmöglichkeiten für Anzeige und Bedienung.

Die Software ist bereit, im Wintersemester 24/25 für die Veranstaltung „Datenstrukturen und Algorithmen“ verwendet zu werden. Sie eignet sich sowohl zum Präsentieren während der Vorlesung, als auch als Material für die Studierenden zum eigenständigen Gebrauch z. B. beim Nachbereiten der Vorlesung.

Der Software weitere Sortierverfahren hinzuzufügen ist, wenn keine weiteren Darstellungselemente benötigt werden, durch Implementieren weiterer Module relativ einfach möglich. Für die Funktionen eines neuen Sortierverfahrens muss das Modul jeweils eine Klasse enthalten, die durch Attribute einen eindeutigen Namen und einen Anzeigenamen festlegt und die Methoden zum Ausführen eines Schritts, zum Speichern und Laden des Zustands, für den Text auf dem Ausführungsstapel und zum Entfernen aus der Darstellung implementiert. Diese Klassen können selbst weitere Listen und Referenzen auf Elemente von Listen erzeugen. Sie können selbst von der Klasse `Pseudocode` erben, und die Methoden zum Festlegen des Texts und der Hervorhebung von Schritten überschreiben, oder ein Objekt einer Klasse erzeugen die von `Pseudocode` erbt und die Methoden überschreibt. Die Klasse der Funktion, die die gesamte Liste sortiert, muss in `Algorithm.algorithms` eingetragen werden, und in ihrem Konstruktor Inhalt und Position der Legende, Sichtbarkeit des Aufrufsstapels und die Anzeige von Namen der Referenzen ober- oder unterhalb der Liste festlegen. Die Klassen aller anderen Funktionen des Verfahrens müssen in `Algorithm.functions` eingetragen werden. Menüpunkte zum Wechsel der Verfahren werden von der Hauptklasse automatisch erzeugt. Denkbar wäre es auch, der Software alternative Varianten der bestehenden Verfahren hinzuzufügen, z. B. Quicksort mit anderer Wahl des Pivotelements, oder Mergesort mit Zählen der Inversionen der Liste. Die zur Darstellung verwendete Klasse `QGraphicsItem` unterstützt auch Interaktionen mit dem Benutzer, es wäre also grundsätzlich möglich eine Version von Quicksort zu implementieren, die dem Benutzer die Wahl des Pivotelements per Mausklick ermöglicht. Vor allem der Inhalt der Software könnte also erweitert werden.

Literatur

[Fou24a] Python Software Foundation. *json — JSON encoder and decoder*. 2024. URL: https://docs.python.org/3/library/json.html (besucht am 21. 05. 2024).

[Fou24b] Python Software Foundation. *tkinter — Python interface to Tcl/Tk*. 2024. URL: https://docs.python.org/3/library/tkinter.html (besucht am 24. 05. 2024).

[Hoa61a] C. A. R. Hoare. „Algorithm 63: Partition“. In: *Commun. ACM* 4.7 (1961), S. 321. DOI: 10.1145/366622.366642.

[Hoa61b] C. A. R. Hoare. „Algorithm 64: Quicksort“. In: *Commun. ACM* 4.7 (1961), S. 321. DOI: 10.1145/366622.366644.

[Hoa62] C. A. R. Hoare. „Quicksort“. In: *Comput. J.* 5.1 (1962), S. 10–15. DOI: 10.1093/COMJNL/5.1.10.

[Knu98] Donald Ervin Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd Edition*. Addison-Wesley, 1998. ISBN: 0201896850. URL: https://www.worldcat.org/oclc/312994415.

[Lim] Riverbank Computing Limited. *What is PyQt?* URL: https://www.riverbankcomputing.com/software/pyqt/intro (besucht am 24. 05. 2024).

[Ltd24] The Qt Company Ltd. *Qt for Python*. 2024. URL: https://doc.qt.io/qtforpython-6/ (besucht am 24. 05. 2024).

[Tea] The GTK Team. *GTK and Python*. URL: https://www.gtk.org/docs/language-bindings/python/ (besucht am 21. 05. 2024).