Übung 1 - Algorithmen & Daten Strukturen

1 Erarbeitung einer einfach verketteten Liste.

Eine einfach verkettete Liste besteht aus Knoten denen ihr Nachfolge Knoten bekannt ist, die Liste ist demnach eine sequentielle Speicherung von Knoten. Die Knoten selber enthalten letztendlich die Nutzlast der Datenstruktur Liste.
Zur Visualisierung der Idee dient folgende Illustration, aus dieser ist zu entnehmen, dass der Letzte Knoten in der Liste keinen Nachfolger besitzt.

Diese Eigenschaft wird in der Implementierung genutzt das Ende der Liste zu identifizieren.

Abbildung 1Datenstruktur einer Liste

Vorteile gegenüber einem Array:

- Eine Liste ist dynamisch, weitere Knoten können jeder Zeit hinzugefügt werden
- Einfügen und Löschen von Datensätzen ist zu mindestens am Anfang und am Ender der List in konstanter Zeit möglich.

Nachteile gegenüber einem Array:

- Ein direkter Zugriff auf ein Element ist nicht möglich, für den Zugriff auf einen Knoten ist es erforderlich erst über seine Vorgänger zu iterieren um auf den gewünschten Knoten zu greifen zu können.

1.1 Entwerfen Sie die notwendigen Datentypen und begründen Sie gegebenenfalls Ihre Wahl.

Der geforderte Datentyp lässt sich in 3 Datentypen aufsplittern, Liste (List), Knoten (Node) und Student, es wurde sich bewusst dagegen entschieden den Datentyp Student statisch an den Datentyp Knoten zu binden, da dies Veränderungen an der Implementierung bedarf sobald ein neuer Datentyp in der Liste gespeichert werden soll. Daher hat man sich entschieden im Datentyp Knoten einen generischen Zeiger für die Daten zu definieren, um so jeden erdenglichen Datentyp in der Liste speichern zu können. Dies bringt folgende Vorteile mit sich:

- Flexibilität der Implementierung.
- Wiederverwendbarkeit der Datentypen.
- Einfache Schnittstellen zu den Datentypen.
- Unterstützung für die Speicherung verschiedener Datentypen.

Nachteile dieses Ansatzes sind der erhörter Speicherbedarf für die erzeugten Datentypen, dies ist vernachlässigbar da dies keine Anforderung dieser Übung ist.

Weiterhin ist anzumerken, dass der Datentyp List so designt bzw. implementiert wurde, dass eine einfache und eine doppelt verkettete Liste über die gleiche Schnittstelle (Funktionen) zu bedienen ist. Daher muss ein Nutzer der Liste, beim Erstellen der Liste angeben ob er eine einfach oder eine doppelt verkettete Liste wünscht. Dies impliziert natürlich, dass sich Listen und Knoten in Abhängigkeit des gewünschten Listen Typen Verhalten.

Alle drei Datentypen abstrahieren die eigentliche Datenstruktur nach außen hin, durch eine Schnittstelle in Form einfacher Funktionen. Demnach ist die eigentliche Datenstruktur außerhalb der Implementierung ohne Benutzung dieser Funktionen nicht möglich.

Dies bringt folgende Vorteile:

- Vermeidung von inkonsistente Zustände in den Datentypen, die durch Veränderung von außen hervorgerufen werden könnten
- Ermöglichung von Änderungen am Datentyp Quelltext ohne, dass der Consumer-Quellcode der Liste einer Änderung bedarf

1.2 Struktur der Datentypen

Der Datentyp ***Student*** ist folgendermaßen strukturiert:

Der Vor- und Nachname sowie der Name des Studienfaches werden als *Char*-Pointer definiert um Strings beliebiger Größe speichern zu können. Die Matrikelnummer wird als Integer gespeichert und der Präfix `s0` wird der einfachheitshalber nicht mit gespeichert.

Student
char* firstName
char* sureName
char* courseName
int matriculationNumber

Abbildung 2 Datentyp für den Studenten

Der Datentyp ***Knoten*** ist folgendermaßen strukturiert:

Der Knoten einer Liste wird durch zwei Datentypen repräsentiert einen für einfach verkettete Listen und einen für doppelt verketten Listen. Somit kann Speicherplatz gespart werden und die Identifikation des Knotentypen wird über das Attribut „*isDoupleLinkedNode*“ ermöglicht. Somit kann die Liste den Knoten entsprechend behandeln.

NodeDoupleLinked	
bool	isDoupleLinkedNode
void	*data
Node	next
Node	prev

NodeSingleLinked	
bool	isDoupleLinkedNode
void	*data
Node	next

Der Datentyp ***List*** ist folgendermaßen strukturiert:

Die List Datenstruktur selber dient nur als Sockel für die Knoten der Liste daher speichert die Liste den ersten Knoten „*root*“ und den letzten Knoten „*head*“ der Liste. Somit kann in konstanter Laufzeit auf den ersten bzw. den letzten Knoten zugegriffen werden.

Weiterhin wird die Anzahl „*elementCount*“ der aktuellen Knoten in der Liste gespeichert, um einen Zugriff auf die Größe mit konstanter Laufzeit zu gewährleisten.

Der „*destroyDataHandler*“ ist ein optionaler Funktions-Pointer auf eine Funktion welche die Daten eines Knoten deallokieren kann. Aufgrund des Designs der Liste kennt diese den Datentyp der gespeicherten Daten nicht, folglich kann diese die gespeicherten Daten selber nicht deallokieren und muss daher diese Aufgabe an einen „*destroyDataHandler*“ delegieren.

List	
Node	root
Node	head
NodeHandler	destroyDataHandler
size_t	elementCount
bool	isDoupleLinkedList

Für mehr Informationen schauen Sie bitte in die mit DoxyGen generierte Dokumentation

1.3 Implementieren Sie die notwendigen Datentypen in C unter Verwendung von typedef, struct, enum und den entsprechenden gewählten Datentypen.

Siehe den beiliegenden Quellcode bzw. die mit Doxygen generierte Dokumentation.

2 Funktionalität zur Verarbeitung von Elementen einer einfach verketteten Liste

Siehe den beiliegenden Quellcode bzw. die mit Doxygen generierte Dokumentation.

3 Einige Methoden obiger einfach verketteter Liste lassen sich (im Gegensatz zum Array oder einer doppelt verketteten Liste) effizient (in unterschiedlicher Hinsicht) implementieren, andere nicht unbedingt - welche sind das und warum?

(a) Implementieren Sie eine Funktion zum Erstellen eines neuen Elements.

Das Erstellen eines Datensatzes (Element) ist von der Implementierung der Liste getrennt zu betrachten da, dies nicht in der Verantwortung der Liste oder des Arrays liegt.

Daher betrachte ich hier nur die Methoden zum Erstellen eines Knotens für eine Liste. Die Implementierungen dieser Methode unterscheiden sich nicht von der für doppelt verketteten Listen. Lediglich der erstellte Knoten erfordert bei einer doppelt verketten Listen mehr Speicher.

(b) Implementieren Sie eine Funktion zum Hinzufügen eines Elements vor dem ersten Element.

Diese Methode lässt sich für Listen generell in konstanter Zeit Implementieren, da der neue Knoten und der erste Knoten nur über Pointer mit einander verbunden werden.

Die Implementierung für ein Array der Größe N ist wesentlich aufwändiger, da N - 1 Verschiebungen nötig sind um Platz für den neuen Datensatz zu schaffen und das Array möglicherweise erst mittels Rellokierung Vergrößert werden muss.
Daher lässt sich die diese Methode für Listen effektiver als für Arrays implementieren.
Wobei generell zu sagen ist das Arrays weniger Speicher für jeden Datensatz in Anspruch nehmen.

(c) Implementieren Sie eine Funktion zum Hinzufügen eines Elements nach dem letzten Element.

Diese Methode lässt sich für Listen der Größe N generell in konstanter Zeit Implementieren, da der neue Knoten (enthält das neue Element) und der letzte Knoten nur über Pointer mit einander verbunden werden. Dies setzt aber voraus, dass der letzte Knoten bereits bekannt ist andernfalls muss dieser in N-1 Schritten ermittelt werden.

Um diesen Flaschenhals zu umgehen kann der letzte Knoten zwischen gespeichert werden, dies erfordert zwar mehr Speicher und etwas mehr Aufwand bei der Implementierung um den Knoten aktuell zuhalten, bringt aber eine konstante Laufzeit.

Die Implementierung für ein Array der Größe N erfordert die Vergrößerung des Arrays und die damit verbundenen Verschiebe Operationen.

Damit lässt sich diese Methode für Listen im Sinne der Laufzeit effektiver implementieren. Wobei generell zu sagen ist das Arrays weniger Speicher für jeden Datensatz in Anspruch nehmen.

(d) Implementieren Sie eine Funktion zur Ausgabe eines Elements der Liste.

Der Aufwand für das Ausgaben eines Elementes hängt generell vom Datensatz ab und ist unabhängig von davon ob der Datensatz sich in einem Array bzw. in einer Liste befindet. Der eigentliche Unterschied zwischen der Implementierung für eine Array und eine Liste besteht in der Adressierung eines Datensatzes.

Dies ist in einem Array in konstanter Zeit möglich und daher lässt sich diese Methode im Sinne der Laufzeit effektiver implementieren als für eine Liste. In einer Liste muss das gewünschte Element erst durch Iterationen bzw. Suchen ermittelt werden bevor es auf dem Bildschirm ausgeben werden kann.

Somit ist die Implementierung für Arrays im Sinne der Laufzeit effektiver.

(e) Implementieren Sie eine Funktion zur Ausgabe der gesamten Liste.

Für die Ausgabe der Elemente gilt das Gleiche wie für Methode (d). Der entscheidende Faktor ist hier die Iteration über alle Elemente und die entsprechende Implementierung für Listen und Arrays unterscheidet sich hier nicht groß in der Laufzeit. Der Vorteil von Listen gegenüber Arrays besteht darin, dass die Größe der Liste nicht bekannt sein muss, um über alle Elemente zu iterieren.

(f) Implementieren Sie eine Funktion zur Ausgabe der Anzahl der Elemente.

Die Ermittlung der Größe einer Liste erfordert das Zählen von Knoten in der Liste und erfordert demnach N Operationen. Dieser Flaschenhals lässt sich ähnlich wie in Methode (c) umgehen in dem die Größe zwischen gespeichert wird. Was wiederum zu einer konstanten Laufzeit führt und zu einem höheren Speicherverbrauch pro Liste führt.

In einem Array lässt sich die Größe nachträglich nicht mehr ermitteln und muss zwischen gespeichert werden. Somit ist die Laufzeit wie für Listen ebenfalls in konstanter Zeit möglich.

Demnach lässt sich diese Methode "Out of the Box" für Arrays effektiver implementieren. Mit der oben beschrieben Verbesserung unterscheiden sich die Implementierungen für Listen und Arrays hinsichtlich der Laufzeit nicht.

(g) Implementieren Sie eine Funktion zum Löschen eines Elements.

Diese Methode lässt sich in doppelt verketteten Listen hinsichtlich der Laufzeit am effektivsten implementieren. Da beim Löschen eines Elements aus der Liste der Vorgänger des Elementes bekannt sein muss um den Vorgänger mit dem Nachfolger zu verbinden. In einfach verketteten Listen muss der Vorgänger erst mittels Iteration ermittelt werden, in doppelt verketteten Listen ist dies nicht erforderlich da die Adresse des Vorgänger Knoten im zu löschenden Knoten zwischen gespeichert ist.

Die Datensätze eines Arrays lassen sich in konstanter Zeit löschen in dem der Platz des Arrays als gelöscht markiert wird. Ist die nicht möglich muss das Array verkleinert werden und die Elemente verschoben werden.

(h) *Implementieren Sie eine Funktion zum Löschen der gesamten Liste.*

Die Implementierungen für einfach und doppelt verkettete Listen unterscheiden sich hinsichtlich der Laufzeit nicht. Da in beiden Fällen über jeden Knoten iteriert werden müssen um dessen Nutzlast und den Knoten selbst zu löschen bzw. zu deallokieren.

Die Implementierung für eine Array erfordert im Grunde nur die Deallokierung vom Array selbst. Sind im Array Datensätze gespeichert die einer Deallokierung bedürfen ist zusätzlich eine Iteration über das ganze Array im Vorfeld erforderlich.

Demnach lässt sich die diese Methode für Arrays hinsichtlich der Laufzeit effektiver implementieren.

(i) Implementieren Sie Funktionalität zum Suchen eines oder mehrerer Studenten nach Vor- und Nachname, Matrikelnummer und Studiengang.

Da Arrays im Gegensatz zu Listen einen wahlfreien Zugriff in konstanter Zeit auf die Datensätze gewähren, lässt sich dies mittels binärer Suche effektiver für sortierte Arrays implementieren.

Demnach ist ein Aufwand von O(log n) nötig. In sortieren Listen ist ein wesentlich höherer Aufwand zu erwarten da für jeden Durchlauf der Binary Search die Iteration zum aktuellen Mittel Knoten erforderlich ist bzw. da Listen keinen wahlfreien Zugriff in Konstanter Zeit unterstützen.

(j) Implementieren Sie Funktionalität zum Sortieren der Datensätze der Studenten, Matrikelnummer und Studiengang nach zwei selbstgewählten Sortierverfahren.

Generell lassen sich Sortierverfahren, welche einen direkten Zugriff auf die Elemente erfordern, für Arrays effizienter hinsichtlich der Laufzeit implementieren im Gegensatz zu Listen.

Da Listen im Gegensatz zu Arrays keinen Wahlfreien-Zugriff in konstanter Zeit ermöglichen. Ein Beispiel hierfür ist Quick-Sort dieser „Divide and Conquer“ Algorithmus teilt die Datensätze jeweils in zwei Teile die durch ein Pivot Element voneinander getrennt sind. Die Ermittlung dieses Pivot Element ist in Arrays durch eine einfache Division zu ermitteln, in einer Liste muss zusätzlich noch über Knoten iteriert werden um das Pivot Element zu ermitteln.

Im Conquer Schritt sind ebenfalls zusätzliche Iterationsschritte nötig damit Elemente die kleiner sind als das Pivot Element in den Linken Teil des Datensatzes Verschoben werden und die die größeren in den rechten Teil. Somit kann man solche Algorithmen praktisch für Arrays hinsichtlich der Laufzeit effizienter implementieren.

Ein Sortierverfahren wie Merge-Sort unterstützt aber dafür das Sortieren von Listen, welche nicht komplett in den Speicher passen, für solche Anwendungsfälle eignen sich Listen erheblich besser. Da Merge-Sort ein "Out of place" verfahren ist benötigt es für Arrays mehr Speicher um dieses zu sortieren.
Für Listen existiert dieser Nachteil je nach Implementierung nicht. Somit eignet sich Merge-Sort für Listen wesentlich besser als für Arrays. In dieser vorliegenden Implementierung benötig Merge-Sort etwas mehr Speicher, da im Sockel der Liste zusätzliche Informationen gespeichert werden (bspw. Anzahl der Knoten, Adressen für den ersten und letzten Knoten).

Aber dafür wird man mit einem theoretischen Aufwand von O(n * log(n)) belohnt.

4 Implementieren Sie obige Datenstruktur und 4 der oben genannten Funktionalitäten (möglichst laufzeiteffizienter) als doppelt verkettete Liste.

Da die Liste beide Varianten von vorn herein Unterstützt und deren einziger Unterschied in der Ermittlung des Vorgänger Knoten besteht Ist dies bereits implementiert worden.

Für weitere Informationen, siehe den beiliegenden Quellcode bzw. die mit Doxygen generierte Dokumentation.

5 Analysieren Sie die Komplexität der von ihnen implementierten Sortierverfahren allgemein und im speziellen Fall Ihrer Implementierung.

Es wurde sich für die Sortierverfahren Bubble- und Merge-Sort entschieden, da ersteres einfach zu implementieren ist und letzteres sich gut für verkettete Listen eignet.

5.1 Bubble-Sort

Im Folgenden wollen wir uns der Komplexität von Bubble-Sort widmen und zwar erst im allgemeinen Fall und im speziellen Fall der vorliegenden Implementierung.
Im allgemeinen Fall wird die Eingabe-Liste der Größe *N* von links nach rechts durchlaufen. Dabei wird in jedem Schritt das aktuelle Element mit seinem Nachfolger verglichen. Falls dieses Element größer/kleiner ist als sein Nachfolger werden sie vertauscht. Am Ende der Phase steht bei auf- bzw. absteigender Sortierung das größte bzw. kleinste Element der Eingabe am Ende der Liste. Die Bubble-Phase wird solange wiederholt, bis die Eingabeliste vollständig sortiert ist. Der günstigste Fall liegt vor sofern die Liste bereits sortiert vorliegt, der schlechteste Fall liegt vor sofern die Liste in umgekehrter Reihenfolge vorliegt.

Sowohl für den schlechtesten und den besten Fall sind $(N-1)$ $(N-1)$ $\times (Vergleiche +$ Vertauschungen) nötig um die Liste zu sortieren. Somit ergibt sich ein Aufwand von $(N^2 - 2N + 1) \times 2$ Operationen was einer quadratischen Laufzeit bzw. $\mathrm{O}(N^2)$ führt.

Der hier beschriebene allgemeine Fall ignoriert die Tatsache, dass das letzte Element des vorherigen Durchlaufs nicht mehr betrachtet werden muss, da die restliche zu sortierende Liste keine größeren bzw. kleineren Elemente mehr enthält. Weiterhin kann verhindert werden, dass bereits sortierte Listen erneut sortiert werden, in dem nach jedem Durchlauf geprüft wird ob überhaupt eine Vertauschung stattfand, ist dies nicht der Fall ist die Liste bereits sortiert und die Sortierung kann beendet werden. Diese beiden Tatsachen werden in der hier vorliegenden Implementierung beachtet und somit ergeben sich für den Speziellen Fall folgende Laufzeiten.

Für den günstigsten Fall ergibt sich so ein Aufwand von $(N-1) \times Vergleichen +$ $0\ Vertauschungen$ bzw. $\mathrm{O}(N)$. Der Grund hierfür liegt in der Tatsache, dass eine bereits sortierte Liste auch nur einmal durchlaufen werden muss bevor der Algorithmus terminiert.

Im schlechtesten Fall muss die komplette Liste erneute sortiert werden dies heißt das
Pro durchlauf das größte bzw. kleinste Element ans Ende der Liste geschoben wird somit

Kann bei jeden Durchlauf ein weiterer Vergleich eingespart werden. So sind im ersten Durchlauf noch $n-1$ Vergleiche und Vertauschungen nötig so sind es im 2ten Durchlauf nur noch n – 2 Vergleiche und Vertauschungen und so weiter.
Die Nummer des Durchlaufs wollen wir an dieser Stelle mal als $i \in \mathbb{N}$ bezeichnen wobei n $\in \mathbb{N}$ die Anzahl der Elemente der Liste ist, somit lässt sich mit folgender Summenformel $\sum_{i=1}^{n-1}(n-i) = (n-1)+(n-2) \ldots 1$ die Anzahl der nötigen Vergleiche und Vertauschungen berechnen. Diese Summenformel lässt sich leicht über den „Kleinen Gauß“ herleiten in dem man $n-2$ für n einsetzt, sobald man sich die Zahlenfolge aufschreibt wird dies ersichtlich da wir im ersten Durchlauf $n-1$ Vergleiche / Vertauschungen haben und insgesamt nur $n-1$ Durchläufe haben.

$$\frac{(n-2)\times(n-2+1)}{2} \Leftrightarrow \frac{(n-1)\times(n-2)}{2} \Leftrightarrow \frac{n^2-3n+2}{2}$$

Mit dieser abgewandelten Summenformel können wir die Anzahl von Operationen wie folgt berechnen: $\frac{n^2-3n+2}{2} \times (1 \times Vertauschung + 1 \times Vergleich) \Leftrightarrow 2 \times \frac{n^2-3n+2}{2}$ somit ergibt sich für den schlechtesten Fall eine Quadratische Laufzeit.

Im mittleren Fall errechnet sich in dem man aus der Anzahl der Operationen für den günstigsten und den schlechtesten Fall den Mittelwert bildet und Summanden mit dem nicht maximalen Exponenten entfernt und die Konstanten des Summanden mit maximalen Exponenten entfernt:

$$\frac{n-1+2\times\frac{1}{2}\times n^2-3n+2}{2} \Leftrightarrow \frac{n^2-\cancel{2n}+\cancel{1}}{\cancel{2}} \Longrightarrow \mathcal{O}(n^2)$$

Somit unterscheidet sich die Komplexität der vorliegende Implementierung im Mittleren Fall nicht welcher dort auch einer Quadratischen Laufzeit entspricht. Lediglich der günstigste Fall besitzt eine lineare Laufzeit. Die ist keine wirkliche Verbesserung im Gegensatz zur allgemeinen Implementierung daher gehört Bubble-Sort nicht zu den schnellsten Sortierverfahren.

5.2 Merge-Sort

Merge-Sort ist ein Divide-Conquer-Algorithmus welcher aus zwei Schritten besteht zum einem aus dem Divide-Schritt welcher die Liste in der Mitte in zwei Teillisten teilt und somit das Problem in Teilprobleme teilt. Die Eingabe Liste wird rekursiv solange in Teilprobleme geteilt solange deren Größe $n > 1$ ist. Der zweite Schritt besteht darin die entstanden Teilprobleme zu lösen und zu einer Gesamtlösung zu kombinieren, im Falle von Mergesort werden die Teillisten in korrekter Reihenfolge wieder zusammen geführt.
Wie bereits erwähnt handelt es sich bei diesem Verfahren um ein Divide & Conquer Algorithmus, dessen Komplexität lässt sich allgemein wie folgt definieren:

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ b \times T\left(\frac{n}{b}\right) + D(n) + C(n), & n > 1 \end{cases}$$

$b \coloneqq Anzahl\ der\ Teilprobleme$
$n \coloneqq Größe\ der\ Eingabeliste$
$D(n) \coloneqq Anzahl\ der\ Operationen\ zum\ Aufteilen\ des\ Problems\ in\ Teilösungen$
$C(n) \coloneqq Anzahl\ der\ Operationen\ zum\ Kombinieren\ der\ Teilösungen\ zur\ Endlösung$

Die Effizienz für Merge-Sort kann demnach wie folgt definiert werden, da der Divide-Schritt die Eingabe Liste jeweils in zwei Teilprobleme teilt ist für $b = 2$ einzusetzen. Der Divide-Schritt kann im Allgemeinen in konstanter Zeit berechnet werden da dieser nur die Mitte der Sub Liste berechnet.

Daher ist $D(n) = \Theta(1)$. Der Merge-Schritt welche die Teillisten zu einer Gesamtliste zusammenführt ist in Linearer-Zeit möglich, da dieser nur die Elemente der Teillisten in die Gesamtliste verschiebt. Daher ist $D(n) = \Theta(n)$. Somit beträgt die Komplexität dieses Verfahren im Allgemeinen Fall also

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2 \times T\left(\frac{n}{2}\right) + \Theta(n), & n > 1 \end{cases}$$

Stellt man sich die aufgeteilte Eingabeliste in einen Baum vor so erkennt man, dass der Merge-Schritt je Ebene $C(n)$ Schritte benötigt. Daraus folgt, dass der Algorithmus insgesamt $C(n) \times \log(n) + C(n)$ Schritte benötigt. Somit ergibt sich für den Allgemeinen Fall eine Komplexität von

$$\mathcal{O}(n * \log(n))$$

Da Merge-Sort sich für den günstigsten Fall (die Liste ist bereits sortiert) und den schlechtesten Fall (die Liste liegt in umgekehrter Reihenfolge vor) im Sinne der Komplexität identisch verhält muss dieser hier nicht gesondert betrachtet werden.

Die vorliegende Implementierung für verkettete Listen unterscheidet sich lediglich nur im Divide-Schritt, da hier die Ermittlung der Mitte alleine nicht ausreichend ist, da jeweils noch zum Mittelknoten iteriert werden muss. Daher wollen wir uns an dieser Stelle mit der Analyse für die Komplexität für verkettete Liste widmen. Dazu schauen wir uns den Divide-Schritt genauer an, dieser benötigt in Abhängigkeit der Größe der Teilliste, die wir hier mit sn bezeichnen wollen, $\frac{sn}{2}$ Schritte mehr pro Divide-Aufruf-Tiefe bzw. pro Ebene des gedachten Binär-Baums. Somit ist unsere Formel wie folgt anzupassen:

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2 \times T\left(\frac{n}{2}\right) + \frac{n}{2} \times \frac{\log(n)}{\log(2)} + \Theta(n), & n > 1 \end{cases}$$

Kürzen wir jetzt alle von n unabhängigen Konstanten und die variablen mit den geringsten Exponenten so erhalten wir:

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2 \times T\left(\frac{n}{2}\right) + n \times \log(\mathrm{n}) + \Theta(n), & n > 1 \end{cases}$$

Und nach dem wir $2 \times T\left(\frac{n}{2}\right)$ aufgelöst haben erhalten wir:

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ n \times log(n) + n \times \log(\mathrm{n}) + \Theta(n), & n > 1 \end{cases}$$

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2 \times (n \times log(n)) + \Theta(n), & n > 1 \end{cases} \Rightarrow \mathcal{O}(n * \log(n))$$

Somit ist ersichtlich das diese Implementierung auch eine Logarithmische Laufzeit aufweist, somit hat diese Implementierung eine Laufzeit $\mathcal{O}(n * \log(n))$. Die Differenzierung der Fälle ist an dieser Stelle nicht nötig, da Merge-Sort zum den unstabilen Sortierverfahren gehört, daher sortiert es auch bereits korrekte sortierte Elemente erneut. Weiterhin ist kein Best-Case auszumachen, dies erklärt sich daher, dass Merge-Sort ein Worst-Case optimiertes Verfahren ist.