HAW HAMBURG

ADP01 Aufgabe 2 Team 7

Sortieralgorithmen Entwurf

Soheil Samar, Tjark Pfeiffer

10.11.2025

Inhaltsverzeichnis

AD Sortieralgorithmen Entwurf

1. Einleitung

In dieser Aufgabe sollen wir zunächst Entwürfe zu verschiedenen Sortieralgorithmen entwickeln und diese anschließend in Erlang implementieren und ihre Laufzeiten miteinander vergleichen. Dieser Entwurf soll die Kernkonzepte, Anforderungen, sowie die zu nutzenden Datenstrukturen darstellen. Bei den vier Algorithmen handelt es sich um Insertionsort, Radixsort, Introsort und Heapsort.

2. Anforderungen

2.1 Funktional

Kernkonzept der Sortierungsalgorithmen im Entwurf mit Flussdiagrammen und mit Pseudocode darstellen.

Die Sortieralgorithmen müssen die gegebene Liste von Zahlen in aufsteigender Reihenfolge korrekt sortieren.
Die Laufzeiten sollen wie folgt aussehen:
Insertionsort: O(n^2)
Radixsort: O(n*l) (l = maximale Anzahl an Digits)
Heapsort: O(n* log2(n))

- Flussdiagramme für die Algorithmen
- Mit Pseudocode
- Keine Arrays

2.2 Technisch

Implementierung der Sortierungsalgorithmen in Erlang, welche auf Listen arbeiten.

- Implementierung in Erlang
- Keine vorgefertigten Hilfsfunktionen verwenden
- Vorgegebene Schnittstellen:
    - insertionS:insertionS(<Liste>)
    - radixS:radixS(<Liste>,<Digit>)
    - introS:introS(<pivot-methode>,<Liste>,<switch-num>)
    - heapS:heapS(<Liste>)
        - heap:create/0
        - heap:isEmpty/1
        - heap:insert/2, heap:insert(<Heap>,<Element>)
        - heap:pop/1
        - heap:top/1

3. Datenstrukturen

Als Datenstruktur für die Sortieralgorithmen verwenden wir Listen, um hier den geordneten Aufbau von Listen zu verwenden, damit wir die Elemente darin sortieren können. Außer für Heapsort, für welchen ein eigene Datenstruktur implementiert werden soll.

Insertionsort:

Für die Implementierung ist die Verwendung von zwei Listen geeignet, die jeweils einen sortierten Bereich sowie einen unsortierten Bereich von Zahlen darstellen.

Dadurch kann leicht ein Element aus der unsortierten Liste entfernt werden und anschließend in die sortierte Liste eingefügt werden.

Radixsort:

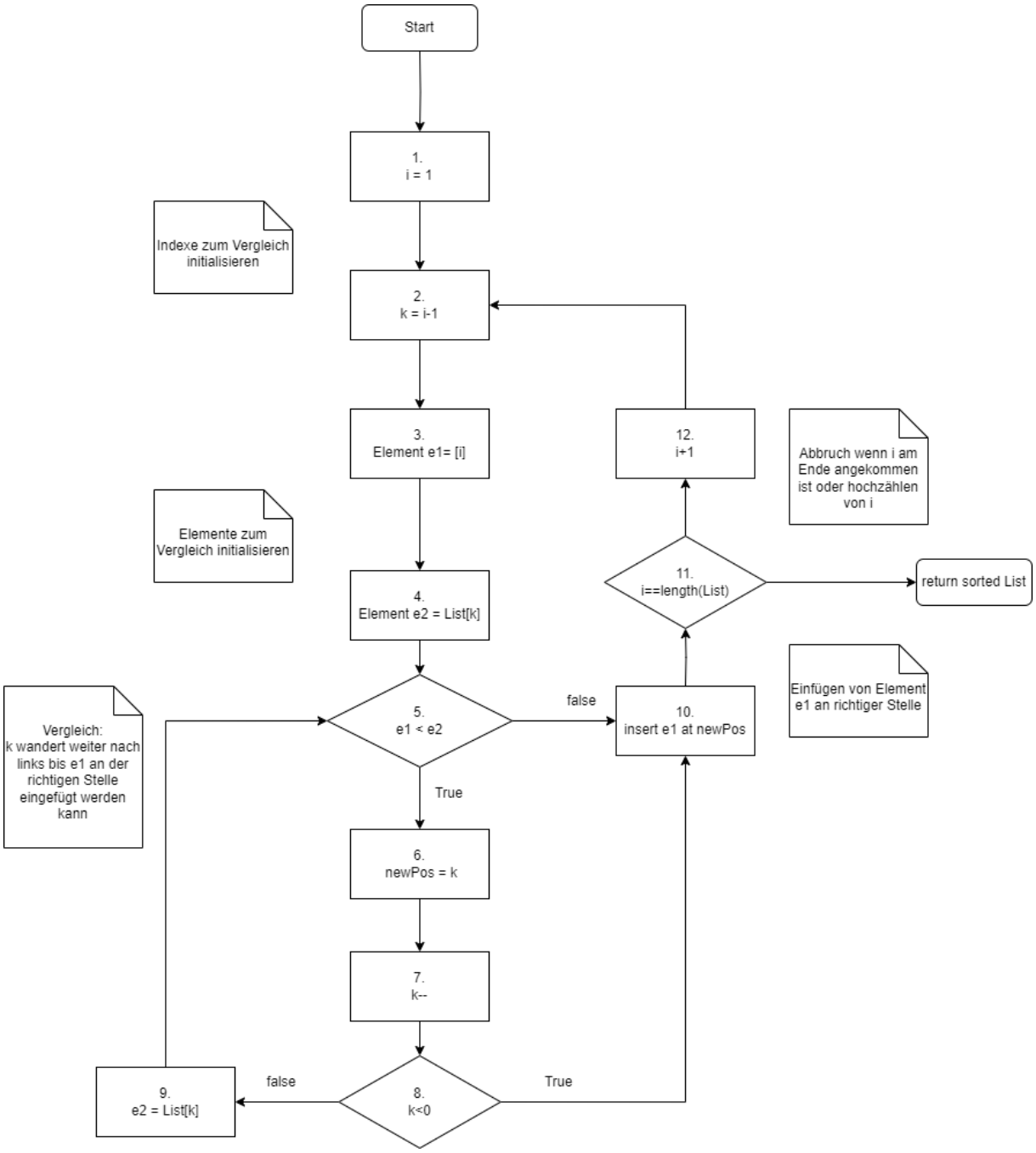
Für die Implementierung sind zwei Listen geeignet: eine Liste, die die Schritt für Schritt sortierten Zahlen beinhaltet und eine Liste mit 10 Elementen, welche selbst Listen sind. Diese 10 Elemente stellen die 10 Eimer dar, auf die die Zahlen je nach Digit verteilt werden.

Heapsort:

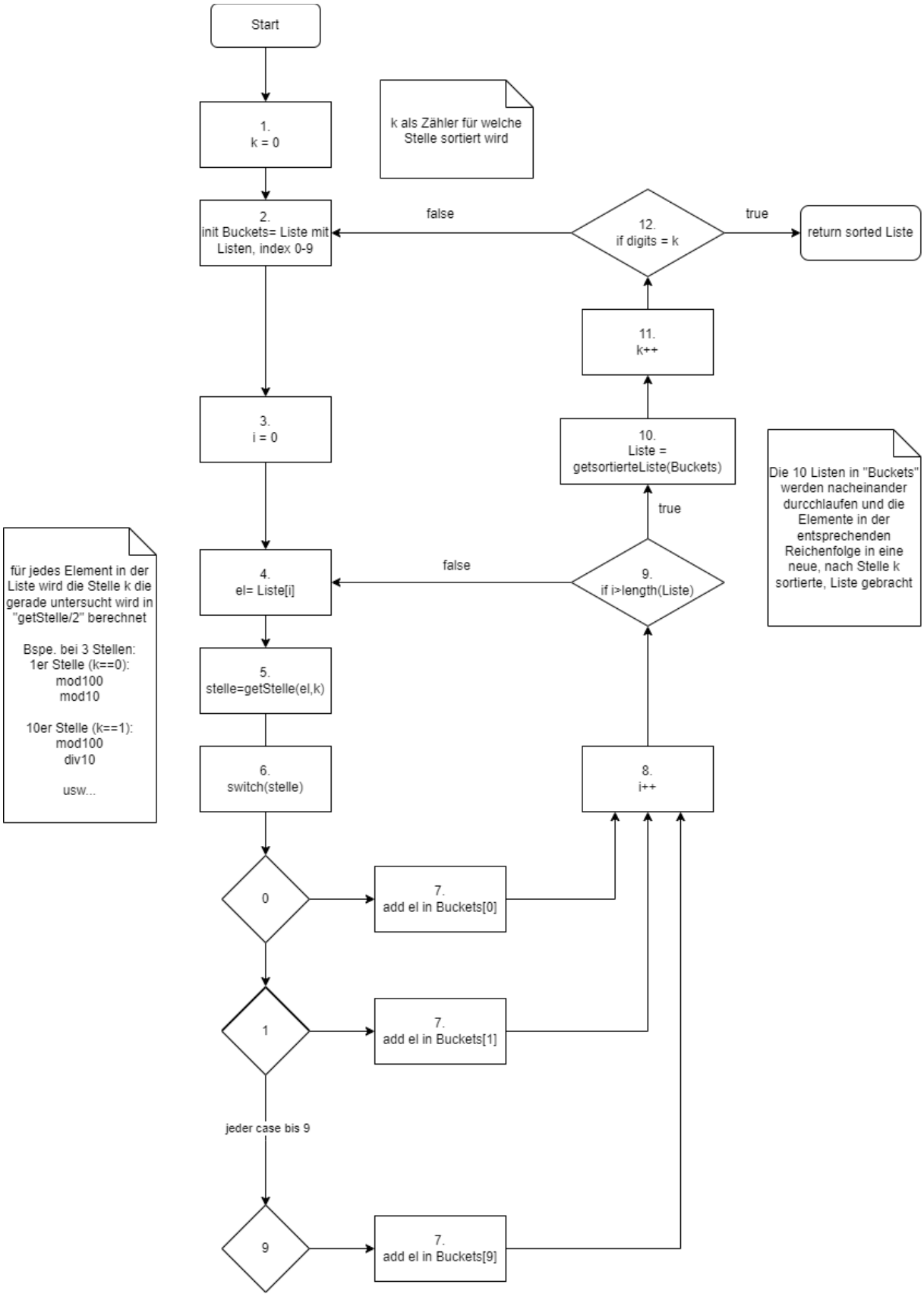
Für die Implementierung ist eine Liste geeignet, welche den sortierten Bereich darstellt, sowie ein Binärbaum, welcher den Max-Heap darstellt. Dadurch kann leicht das Wurzelelement des Max-Heaps als größte Zahl in den sortierten Bereich überführt werden. Außerdem bietet die Struktur des Binärbaums eine logarithmische Laufzeit an, wenn ein Element im Baum gesucht wird.

4. Detaillierte Algorithmen

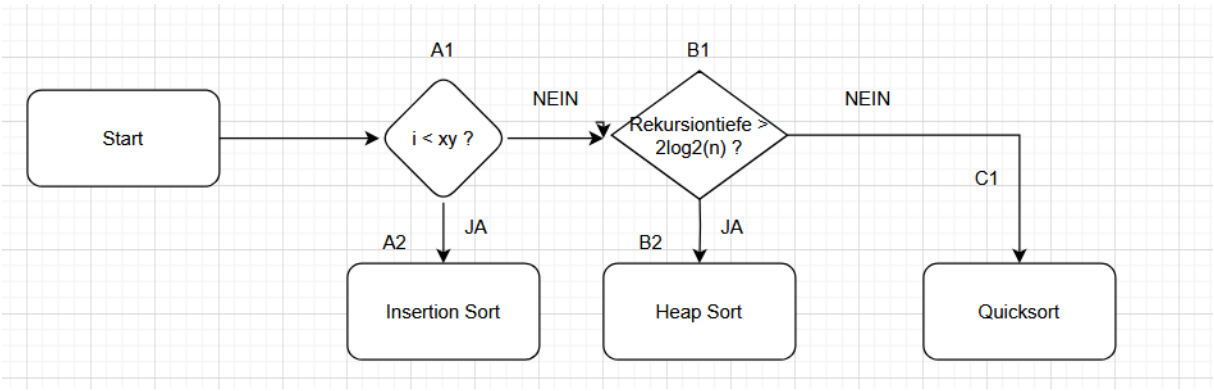
4.1 Insertionsort



4.2 Radixsort

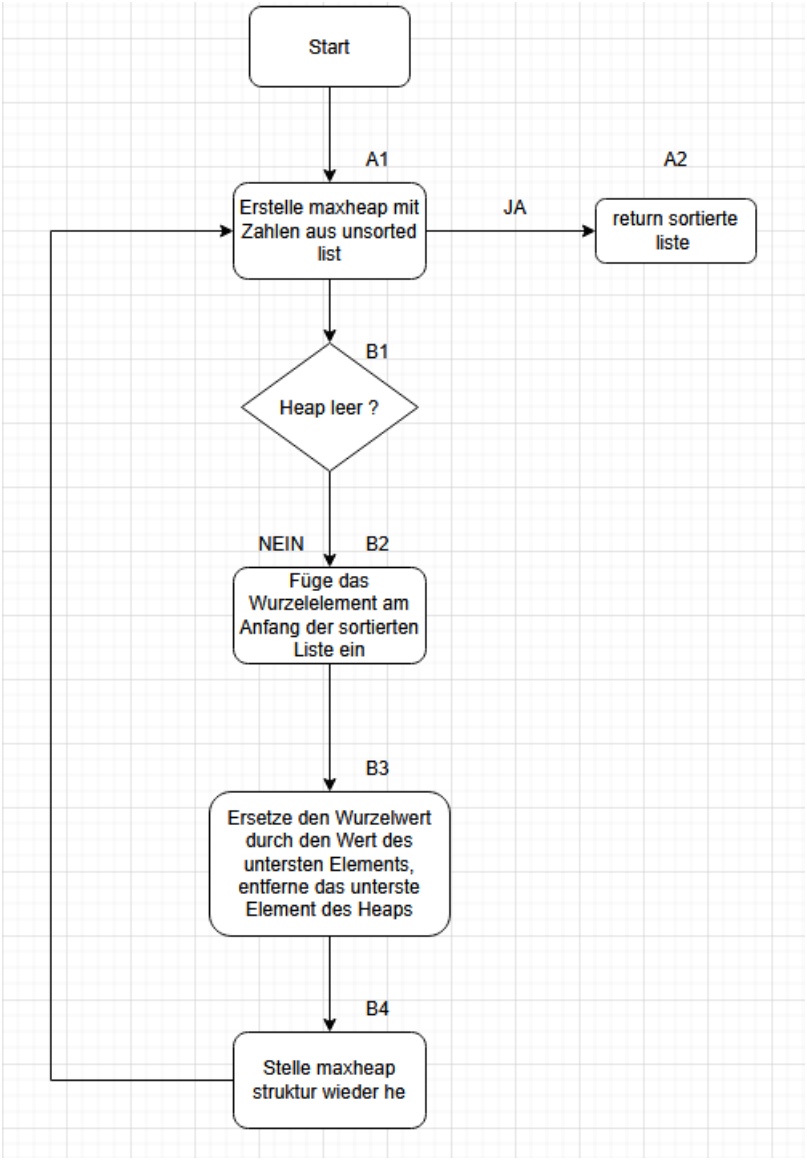


4.3 Introsort



Erwartete Laufzeit: Mit kleinem n: O(n^2) und mit größerem n logarithmischen

4.4 Heapsort



5. Testplan

Testfall	Eingabe	Erwartete Ausgabe	Bemerkung
1.	[]	[]	Leere Liste
2.	[55]	[55]	Ein Element
3.	[1,2,3,4,5]	[1,2,3,4,5] (Bei InsertionS sehr langsam)	Aufsteigend sortiert
4.	[5,4,3,2,1]	[1,2,3,4,5] (Bei InsertionS sehr schnell)	Absteigend sortiert
5.	[9,1,5,7,6,4]	[1,4,5,6,7,9]	Zufällige Werte
6.	[7,7,7,7]	[7,7,7,7]	Gleiche Elemente
7	[3123143563,675474, 746474747]	[675474, 746474747, 3123143563]	Große Zahlen (bei RadixS langsamer)

8.	[-3,-8,-4]	[-8,-4,-3]	Negative Zahlen
9.	Tiefe>2log2(n)	[sortierte Liste]	Bei IntroS: Wechsel zu HeapS
10.	Kleine Liste	[sortierte Liste]	Bei IntroS: Wechsel zu InsertionS

Ziel der Laufzeittests:

Das Hauptziel der Laufzeittests ist es, die Leistung, Schnelligkeit und Effizienz der implementierten Sortieralgorithmen unter verschiedenen Bedingungen zu vergleichen.

Erwartete Ergebnisse:

Wir erwarten, dass Insertionsort bei kleinem n schnell ist und bei großem n sehr langsam wird. Außerdem wird Insertionsort je nach Implementierung bei aufsteigender oder absteigender Vorsortierung entweder sehr schnell oder sehr langsam.

Wir erwarten, dass Radixsort unabhängig von der vorherigen Sortierung gleich schnell sein wird, da alle Digits untereinander verglichen werden. Dazu wird Radixsort langsamer desto höher die maximale Anzahl an Digits ist.

Heapsort wird wahrscheinlich bei geringer Elementanzahl länger brauchen als die naiven Algorithmen, da intern zuerst die Heapstruktur erzeugt wird. Die Erzeugung der Heapstruktur zahlt sich erst bei großem n aus, sodass dort der Heapsort vergleichsweise schnell wird.

Wir vermuten, dass Introsort der insgesamt schnellste Algorithmus wird, da er die bisher genannten Algorithmen (Quicksort statt Radixsort) kombiniert und sie dann einsetzt, wenn ihre Laufzeit am effizientesten ist.