

Einführung in die Informatik II

28.02. und 02.03.2020

1 Radixsort

Das Sortierverfahren *Radixsort* kann dazu verwendet werden, Mengen von *Schlüsseln* fester Länge aufsteigend zu sortieren. Jeder Schlüssel ist eine Folge von m Stellen.

Zum Sortieren verwendet *Radixsort* k Fächer, je ein Fach pro möglichem Stellenwert.

Beispiele:

Beispiel 1: Die Schlüssel sind Worte, die aus m Kleinbuchstaben (= Stellen) bestehen. Hierbei sind $k = 26$ Fächer anzulegen (ein Fach pro Buchstabe).

Beispiel 2: Als Schlüssel werden Zahlen mit m Ziffern (= Stellen) verwendet. Dies führt zu $k = 10$ Sortierfächern für die Ziffern 0..9.

Abweichend von der Vorlesung implementieren wir die einzelnen Fächer als Keller.

Die Vorgehensweise von *Radixsort* lässt sich wie folgt in Pseudocode darstellen, wobei sich die zu sortierenden n Schlüssel in dem Array `keys` befinden:

```
1 | k Kellerfaecher anlegen
2 |
3 | //Alle Positionen der Schlüssel von hinten durchlaufen
4 | for (pos <- m - 1 to 0 by -1) {
5 |
6 |     //1. Phase: Verteilen des Inhalts von keys auf die Kellerfaecher
7 |     for (i <- 0 to keys.length - 1) {
8 |         Ablegen von keys(i) auf das Kellerfach mit dem Index,
9 |         den man keys(i) an der Stelle pos entnimmt
10 |     }
11 |
12 |     //2. Phase: Leeren der Kellerfaecher und Schreiben nach keys
13 |     for (j <- k - 1 to 0 by -1) {
14 |         Leeren des Kellerfachs mit dem Index j;
15 |         dabei keys von hinten füllen
16 |     }
17 | }
```

Hinweis: Die Position *pos* wird (Scala-üblich) von 0 beginnend gezählt. Mit $pos = 2$ wird daher auf die Einerstelle einer dreistelligen Zahl (also bei $m = 3$) zugegriffen!

- a) Wir verwenden dieses Verfahren auf Zahlen mit maximal $m = 3$ Stellen an. Die Zahlen stehen rechtsbündig im Schlüssel und werden bei Bedarf mit führenden Nullen ergänzt. Hierbei sei `keys` wie folgt initialisiert:
`keys = Array(978, 100, 7, 391, 57, 831, 110, 470, 612, 217)`

Nachfolgend finden Sie den Inhalt der Kellerfächer für die letzte Position ($pos = 2$) nach Abschluss der 1. Phase:

| | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 470 | | | | | | | 217 | | |
| 110 | 831 | | | | | | 57 | | |
| 100 | 391 | 612 | | | | | 7 | 978 | |
| (0) | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) |

Nach dem Übertragen der Kellerfächer (2. Phase) sieht `keys` wie folgt aus: `keys = Array(100, 110, 470, 391, 831, 612, 7, 57, 217, 978)`

Beachten Sie, dass sich sowohl beim „Einkellern“ in Phase 1 als auch bei der Entnahme in Phase 2 die Reihenfolge eines Fachs umkehrt. Nach dieser zweimaligen Umkehrung stehen die Elemente in `keys` wieder in der gleichen Reihenfolge wie beim *Radixsort* aus der Vorlesung.

Tragen Sie nun analog für die Positionen $pos = 1$ und $pos = 0$ jeweils den Wert der Kellerfächer nach Abschluss der 1. Phase und den Wert von `keys` nach Abschluss der 2. Phase ein.

$pos = 1$:

| | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| (0) | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) |

`keys` = _____

$pos = 0$:

| | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| (0) | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) |

`keys` = _____

Für $pos = 1$ ergeben sich folgende Kellerfächer:

$pos = 1$:

| | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| | 217 | | | | | | | | |
| 7 | 612 | | | | | | 978 | | |
| 100 | 110 | | 831 | | 57 | | 470 | | 391 |
| (0) | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) |

Nach dem Übertragen sieht `keys` wie folgt aus: `keys = (100, 7, 110, 612, 217, 831, 57, 470, 978, 391)`

Für $pos = 1$ ergeben sich schließlich folgende Kellerfächer:

$pos = 0$:

| | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| | | | | | | | | | |
| 57 | 110 | | | | | | | | |
| 7 | 100 | 217 | 391 | 470 | | 612 | | 831 | 978 |
| (0) | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) |

`keys` sieht danach so aus: `keys = (7, 57, 100, 110, 217, 391, 470, 612, 831, 978)`

- b) Welcher Komplexitätsklasse lässt sich *Radixsort* zuordnen (m entspricht der Länge der Schlüssel, n ihrer zu sortierenden Anzahl)?

- ☐ $O(n \cdot \log(n))$
☐ $O(m \cdot n)$
☐ $O(m + n)$
☐ $O(n^2)$

Begründen Sie Ihre Antwort kurz in Stichworten.

Für alle Positionen m müssen alle n Schlüssel betrachtet werden. Daher $O(m \cdot n)$.

- c) Zur Realisierung der Kellerfächer wird der folgende Typ `Stack` verwendet:

```

1 //Keller
2 class StackElem(var value : Int, var next : StackElem = null)
3 class Stack(var s : StackElem = null)
4
5 //leeren Stack erstellen
6 def emptyStack : Stack = new Stack
7
8 //Pruefen ob Stack leer
9 def isEmpty(stack : Stack) : Boolean = stack.s == null

```

Implementieren Sie folgende Funktionen und Prozeduren. Lösen Sie, wo geboten, eine `IllegalArgumentException` mit geeignetem Fehlertext aus!

```

1 //Element hinzufuegen
2 def push(stack : Stack, value : Int) : Unit = ...
3
4 //Oberstes Element entfernen
5 def pop(stack : Stack) : Unit = ...
6
7 //Oberstes Element vom Stack liefern
8 def top(stack : Stack) : Int = ...

```

Die Implementierungen sehen wie folgt aus:

```

1 //Element hinzufuegen
2 def push(stack : Stack, value : Int) : Unit = {
3   stack.s = new StackElem(value, stack.s)
4 }
5
6
7 //Oberstes Element entfernen
8 def pop(stack : Stack) : Unit = {

```

```

9   if (stack.s == null) throw new
10      IllegalArgumentException("Stack is empty!")
11   else stack.s = stack.s.next
12 }
13
14 //Oberstes Element vom Stack liefern
15 def top(stack : Stack) : Int = {
16   if (isEmpty(stack)) throw new
17      IllegalArgumentException("Stack is empty!")
18   else return stack.s.value
19 }

```

- d) Definieren Sie eine Prozedur `getNumPos`, welche bei gegebenen m maximalen Stellen pro Schlüssel, an der Stelle pos , den Wert von num zurückgibt. Zum Beispiel soll für $m = 3, pos = 2, num = 216$ das Ergebnis 6 sein.

```

1  def getNumPos(m : Int, num : Int, pos : Int) : Int = {
2    var n = num
3    for (i <- 1 to (m - pos) - 1) n /= 10
4
5    return n % 10
6  }

```

- e) Definieren Sie nun eine Prozedur `radixsort`, welche einen Array `keys : Array[Int]` nach dem *Radixsort*-Verfahren sortiert. Überlegen Sie sich vorab, welche Parameter dieser Prozedur zu übergeben sind.

Das Verfahren erfordert (neben dem zu sortierenden Array) die Definition des Parameters m , der die Anzahl der Stellen der zu sortierenden Zahlen angibt. Die Funktion könnte daher wie folgt implementiert werden:

```

1  def radixsort(m : Int, keys : Array[Int]) : Unit = {
2    // Anzahl der Fächer für Dezimalzahlen
3    // Entspricht der Anzahl der Stellen, die getNumPos
4    // zurückgeben kann
5    val k = 10
6
7    // k Kellerfaecher anlegen
8    val stacks = Array.fill(k)(emptyStack)
9
10   // alle Positionen der Schluessel von hinten durchlaufen
11   for (pos <- m - 1 to 0 by -1) {
12     // 1. Phase: Verteilen des Inhalts von keys auf
13     // die Kellerfaecher
14     for (i <- 0 to keys.length - 1) {
15       val key = keys(i)
16       push(stacks(getNumPos(m, key, pos)), key)
17     }
18
19     var idx = keys.length - 1
20
21     // 2.Phase: Leeren der Kellerfaecher und schreiben
22     // nach keys
23     for (j <- k - 1 to 0 by -1) {

```

```
24         val stack = stacks(j)
25         while (!isEmpty(stack)) {
26             keys(idx) = top(stack)
27             pop(stack)
28             idx -= 1
29         }
30     }
31 }
32 }
```