technische universität dortmund

Fakultät für Informatik
Lehrstuhl 11

Mart Hagedoorn
Guangping Li

Sommersemester 2024
29. April

DAP2 Praktikum – Blatt 4

Abgabe: KW 19

Studienleistung

- Zum Bestehen des Praktikums muss jeder Teilnehmer*innen die folgenden Leistungen erbringen:
  - Es müssen mindestens 50 Prozent der Punkte in den Kurzaufgaben erreicht werden.
  - Es müssen mindestens 50 Prozent der Punkte in den Langaufgaben erreicht werden.
- Im Krankheitsfall kann ein Testat bei Vorlage eines Attests in der folgenden Woche nachgeholt werden.
- Wenn ein Praktikumstermin auf einen Feiertag fällt, müssen Sie sich an einem beliebigen anderen Praktikumstermin in der gleichen Woche testieren lassen.
- **Hinweis:** Notieren Sie sich Ihre Punkte nach jedem Testat! Dies dient der eigenen Kontrolle. (Ihr Punktestand kann Ihnen während des Semesters nicht genannt werden.)

Wichtige Information (im Moodle verfügbar)

- Beachten Sie die Erklärung des **Ablaufs (Blatt A)**.
- Beachten Sie die **Regeln und Hinweise (Blatt R)** in der aktuellsten Version!
- Beachten Sie die **Hilfestellungen (Blatt H)** in der aktuellsten Version!

Langaufgabe 4.1: Max-Heaps und Heap-Select (4 Punkte)

In der Vorlesung haben Sie Max-Heaps kennengelernt, die man als Priority Queue oder zum deterministischen Sortieren in $\mathcal{O}(n \log n)$ Zeit verwenden kann. Dieser Algorithmus aus der Vorlesung wurde in einem *top-down*-Ansatz zur Konstruktion des Heapbaums verwendet. Eine andere Methode um einen Heap zu konstruieren ist ein *bottom-up* Ansatz.

In dieser Aufgabe sollen Sie Heap-Sort Algorithmus implementieren die erst ein Max-Heap konstruirt der den globalen *bottom-up* Ansatz nutzt. Das bedeutet, dass neue Elemente am Ende des Arrays hinzugefügt werden und sich somit unten im Baum befinden. Um also nach jedem neuen Element einen korrekten Baum zu haben, müssen diese wiederholend mit ihren Eltern verglichen und wenn nötig ausgetauscht werden.

Nachdem der Max-Heap konstruiert wurde, wird mit Hilfe des Heapbaums nach dem k größten Element gesucht.

- Implementiere die Methode `maxHeapifyUp(data, i)`, die $\mathcal{O}(\log n)$ Zeit benötigt. Das Argument i ist der Index des Knoten, für den die Heap-Eigenschaft gesetzt werden soll. Da hier der *bottom-up*-Ansatz verwendet wird, sollte `data[0, i-1]` bereits ein gerichteter Heapbaum sein.

  ```
  public static void maxHeapifyUp(int [] data, int i)
  ```

- Implementieren Sie die Methode `buildMaxHeap(data)`, welche mittels `maxHeapifyUp` einen Max-Heap erzeugt. Diese läuft in $\mathcal{O}(n \log n)$ Zeit.

  ```
  public static void buildMaxHeap(int [] data)
  ```

- Implementieren Sie die Methode `extractMax(data,n)`, welche in $\mathcal{O}(\log n)$ Zeit das große Element aus dem Heap entfernt und ausgibt. Danach befinden sich die restlichen Heap-Elemente in `data`$[0, n-2]$, und das entfernte Maximum steht an Position `data`$[n-1]$.

  ```
  public static int extractMax(int [] data, int n)
  ```

- Implementieren Sie die Methode `maxHeapifyDown(data, n)`, die in $\mathcal{O}(\log n)$ ausgeführt wird. Diese Methode stellt immer die Heap-Eigenschaft des Knotens mit dem Index 0 wieder her. Am Ende sollte nur `data[0, n - 1]` einen korrekten Max Heap darstellen.

  ```
  public static int maxHeapifyDown(int [] data, int i, int n)
  ```

- Implementieren Sie die Methode `heapSelect(data, k)`, die mithilfe der vorherigen 4 Methoden das k-große Element in `data` findet und ausgibt. Die Methode darf keine Hilfsarrays verwenden. Der Inhalt von `data` darf beliebig permutiert werden.

  ```
  public static int heapSelect(int [] data, int k)
  ```

Abschließend sollen Sie eine geeignete `main`-Methode schreiben, sodass Ihr Porgramm das eine Ganzzahl n und eine Liste von Ganzzahlen $a_0, a_1, \ldots, a_{n-1}$ via Standard-In (Nach jedem Wert bestätigen Sie mit der Enter-Taste), sowie ein Argument k erhält, und dann den k-großten Wert der Liste ausgibt. Wenn es in der Liste Duplikate gibt, dann brauchen Sie diese nicht besonders zu behandeln:

```
java B4A1.java 3 <Enter>
5 1 2 2 3 4 <Enter>
Input Array: [1, 2, 2, 3, 4]
The 3-smallest element is 2
```

Wie immer können Sie davon ausgehen, dass die Eingabe korrekt ist und müssen den Code nicht um diese Eingabe Robust gestalten.

Langaufgabe 4.2: Zufallsexperimente (4 Punkte)

In den Vorlesungen haben Sie die erwarteten Laufzeiten und die Worst-Case-Laufzeiten von Insertion Sort gesehen. In dieser Übung werden Sie diese theoretischen Ergebnisse verifizieren. Zu diesem Zweck dürfen Sie Lösungen aus früheren Aufgabenstellungen verwenden.

- Implementieren Sie die Methode `insertionSort(numbers)`, die `numbers` mit Insertion Sort sortiert und zurückgibt, wie viele Verschiebungen der Algorithmus durchgeführt hat.

  ```
  public static int insertionSort(int[] numbers)
  ```

- Implementieren Sie die Methode `shufflePermutation(numbers)`, die ohne Hilfsarrays eine beliebige neue Permutation von `numbers` berechnet. Schreiben Sie dazu eine Schleife, die über alle Elemente von `numbers` iteriert und jedes Element mit einem zufällig ausgewählten restierende Element vertauscht.

  Eine Zufallszahl können Sie mit den Java-Bibliotheken `ThreadLocalRandom` oder `Random` generieren. Mehr Informationen finden Sie in der Java-Documentation dieser Bibliotheken.

  ```
  public static void shufflePermutation(int[] numbers)
  ```

- Implementieren Sie die Methode `updatePermutation(numbers, counters)`, die die nichtrekursive Version des Algorithmus von Heap[1] verwendet, um alle möglichen Permutationen von `numbers` zu erzeugen. Ein Aufruf der Methode sollte genau einer Iteration des Heap's Algorithmus entsprechen. Nach genau $(n! - 1)$ Aufrufen der Methode hat das Array `numbers` also alle $n!$ möglichen Perumerationen durchgelaufen.

  Nehme eine Ganzzahl `i := 0` und weil `i < n` testiere ob `counters[i] < i`. Falls ja, wenn `i` gerade ist, verwechsel `numbers[0]` mit `numbers[i]`, wenn `i` ungerade ist, verwechsel `numbers[i]` mit `numbers[counters[i]]`. Dann inkrementiere `counters[i]` mit 1 und jetzt ist die nächste Permutation berechnet, jetzt kann die Methode angeregt werden. Falls nein (also `counters[i] >= i`), dann setze `counters[i] := 0` und `i := 0` und beginne diese Schleife wieder am Anfang.

  Beim erste Anfruf von `updatePermutation` sollte `counters` ein Array der Größe n sein, das mit Nullen gefüllt ist.

  ```
  public static void updatePremutation(int[] numbers, int[] counters)
  ```

Abschließend sollen Sie eine geeignete `main`-Methode schreiben, sodass Ihre Algorithmen die Argumenten n und k erhält. Falls $n \leq 10$ nutz das Algorithmus von Heap, um alle möglichen Permutationen zu berechnen von n Ganzzahlen $(1, \cdots, n)$ und zu sehen, wie viele Verschiebungen notwendig sind, um diese Permutationen zu sortieren. Ermitteln Sie auf diese Weise die durchschnittliche Anzahl der erforderlichen Verschiebungen und wie viele Verschiebungen im schlimmsten Fall erforderlich sind. Stimmen Ihre Ergebnisse mit den theoretischen Ergebnissen aus der Vorlesung überein?

Falls $n > 10$, nützen Sie `shufflePermutation`, um k zufällige Permutationen zu berechnen und diese mit Ihrem Insertion Sort Algorithmus zu sortieren. Zählen Sie die Anzahl der Verschiebungen und geben Sie die durchschnittliche Anzahl der Verschiebungen an.

Beispielen auf Nächste Seite!

[1] https://en.wikipedia.org/wiki/Heap's_algorithm

```
java B4A2.java 10 0 <Enter>
Durchschnittliche Anzahl von Verschiebungen: 22.5
Schlimmste Anzahl von Verschiebungen: 45

java B4A2.java 20 10000 <Enter>
Durchschnittliche Anzahl von Verschiebungen: X
```

Wo `X` ein Zahl in der nähe von das Durchschnittliche Anzahl ist.

TIP: rufe immer insertionSort mit `numbers.clone()` auf, um die Reihenfolge des Heap's Algorithmus nicht zu stören.