technische universität dortmund

Fakultät für Informatik
Lehrstuhl 11

Jonas Ellert
Mart Hagedoorn
Guangping Li

Sommersemester 2022
23. April

DAP2 Praktikum – Blatt 4

Abgabe: ab 2. Mai

Studienleistung

- Zum Bestehen des Praktikums muss jeder Teilnehmer 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: Min-Heaps und Heap-Select (3 Punkte)

In der Vorlesung haben sie Max-Heaps kennengelernt, welche Sie als Priority-Queue oder zum deterministischen Sortieren in $\mathcal{O}(n \log n)$ Zeit nutzen können. Die Vorlesungsmaterialien finden Sie im Moodle-Raum der Vorlesung. Natürlich können Sie auch ein gängiges Lehrbuch (z.B. Ïntroduction to Algorithms"von Cormen et al.) verwenden. In dieser Aufgabe sollen sie *Min*-Heaps implementieren, die analog zu Max-Heaps funktionieren, aber immer das kleinste Element in der Wurzel speichern.

- Implementieren Sie die Methode `minHeapify(data,i,n)`, welche $\mathcal{O}(\log_2 n)$ Zeit braucht. Das Argument n ist die Anzahl der Elemente im Heap, welche nicht unbedingt mit der Länge des Arrays übereinstimmt. Das Argument i ist der Knoten, für welchen Sie die Heap-Eigenschaft herstellen wollen.

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

- Implementieren Sie die Methode `buildMinHeap(data)`, welche mittels `minHeapify` einen Min-Heap erzeugt. Dies funktioniert in $\mathcal{O}(\texttt{data.length} \cdot \log_2(\texttt{data.length}))$ Zeit.

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

- Implementieren Sie die Methode `extractMin(data,n)`, welche in $\mathcal{O}(\log_2 n)$ Zeit das kleinste Element aus dem Heap entfernt und ausgibt. Danach befinden sich die verbleibenden Heap-Elemente in $\texttt{data}[0, n-2]$, und das entfernte Minimum steht an Position $\texttt{data}[n-1]$.

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

- Mithilfe eines Min-Heaps können Sie leicht das *Auswahlproblem* lösen. Dieses fragt nach dem k-kleinsten Element eines Arrays der Länge n (ein klassischer Anwendungsfall des Auswahlproblems ist die Suche nach dem Median, also dem $\left\lfloor \frac{n}{2} \right\rfloor$-kleinsten Element). Implementieren Sie eine Methode `heapSelect(data, k)`, die mithilfe der vorherigen drei Methoden das k-kleinste 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 eine Liste von Ganzzahlen via Standard-In und ein Argument k erhält, und dann den k-kleinsten Wert der Liste ausgibt. Wenn es in der Liste Duplikate gibt, dann brauchen Sie diese nicht besonders zu behandeln: Das 3-kleinste Element von $[1, 2, 2, 2, 4, 5, 5, 7]$ ist 2.

Wie immer sollten Sie bei ungültigen Eingaben passende Fehlermeldungen ausgeben!

Überprüfen Sie die Korrektheit des Ergebnis mit einer geeigneten Assertion!

Langaufgabe 4.2: Quick-Select (3 Punkte)

In Aufgabe 1 haben sie das Auswahlproblem gelöst. Mithilfe eines Heaps finden Sie das k-kleinste Element eines Arrays der Länge n in $\mathcal{O}(n \lg n)$ Zeit. Asymptotisch hätten Sie also genauso viel Zeit gebraucht, wenn Sie das Array einfach vollständig sortiert hätten. In der Vorlesung werden sie einen besseren Algorithmus für das Auswahlproblem kennenlernen, welcher es deterministisch und in $\mathcal{O}(n)$ Zeit löst.

Auf diesem Blatt sollen Sie stattdessen den randomisierten Algorithmus *Quick-Select* implementieren, welcher im Prinzip dem Sortieralgorithmus Quick-Sort entspricht, aber bei der Rekursion immer nur den relevanten Teil weiterverfolgt. Für ein Arrayinterval $A[\ell, r]$ funktioniert dies so:

- Sie wollen das k-kleinste Element in $A[\ell, r]$ finden. Dazu wählen Sie entweder eine zufällige Position $p \in \{\ell, \dots, r\}$ (für die randomisierte Version des Algorithmus) oder einfach $p = \ell$ (für die deterministische Version). Nun partitionieren Sie das Arrayinterval genau wie im Algorithmus Quick-Sort, wobei Sie $q = A[p]$ als Pivot-Element verwenden.
- Beim Partitionieren stellen Sie fest, dass m Elemente kleiner als q sind, sodass anschließend $\forall i \in \{\ell, \dots, \ell + m - 1\} : A[i] < q$ und $\forall i \in \{\ell + m, \dots, r\} : A[i] \geq q$
- Wenn $k \leq m$, dann bestimmen Sie rekursiv das k-kleinste Element in $A[\ell, \ell + m - 1]$. Ansonsten bestimmen Sie rekursiv das $(k - m)$-kleinste Element in $A[\ell + m, r]$.

Die Implementierung sieht wie folgt aus:

- Die Methode `partition` führt die Partitionierung aus und bekommt dieses mal (anders als auf Blatt 2) die Pivot-Position übergeben. Es gilt also $\ell \leq p \leq r$.

  ```
  public static int partition(int [] data, int l, int p, int r)
  ```

- Die Methode `quickSelectFirst` findet rekursiv das k-kleinste Element im Interval $[\ell, r]$, und wählt immer $p = \ell$ als Pivot-Position. Die Version ohne Argumente ℓ und r verleichtert den initialen Aufruf.

  ```
  public static int quickSelectFirst(int [] data, int l, int r, int k)
  public static int quickSelectFirst(int [] data, int k)
  ```

- Die Methode `quickSelectRand` findet rekursiv das k-kleinste Element im Interval $[\ell, r]$, und wählt immer eine zufällige Pivot-Position. Die Version ohne Argumente ℓ und r verleichtert den initialen Aufruf.

  ```
  public static int quickSelectRand(int [] data, int l, int r, int k)
  public static int quickSelectRand(int [] data, int k)
  ```

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

- Ihr Programm bekommt wie gewohnt eine Liste von Ganzzahlen über Standard-In. Außerdem erwartet es zwei Argumente: Das erste Argument ist k; das zweite Argument gibt den zu verwendenden Algorithmus an, und ist entweder `"quickr"` (randomisiertes Quick-Select), `"quickf"` (deterministisches Quick-Select), oder `"heap"` (Heap-Select aus Aufgabe 1). Denken Sie an Fehlermeldungen für ungültige Eingaben.

Überprüfen Sie die Korrektheit des Ergebnis mit einer geeigneten Assertion!

Langaufgabe 4.3: Laufzeitmessung (2 Punkte)

Messen Sie die Laufzeit der Algorithmen auf zufälligen und auf bereits aufsteigend sortierten Sequenzen. Welcher Algorithmus ist wann am schnellsten? Vergrößern Sie die Eingabe, bis Sie einen klaren Unterschied feststellen können.

Weitere Hinweise zur Laufzeitmessung finden Sie auf Blatt 2, Aufgabe 2.4.

Achten Sie darauf, dass Sie nur die Zeit für Select messen, also nicht etwa die Zeit zum Lesen der Eingabe oder zum Prüfen von Assertions!