Algorithmen und Datenstrukturen
Musterlösung 2

Martin Avanzini <martin.avanzini@uibk.ac.at>
Thomas Bauereiß <thomas.bauereiss@uibk.ac.at>
Herbert Jordan <herbert@dps.uibk.ac.at>
René Thiemann <rene.thiemann@uibk.ac.at>

22. März, zur Besprechung am 29. März

Aufgabe 1) Suchverfahren

1. Eine mögliche rekursive Implementierung der binären Suche könnte wie folgt aussehen:

Beispiel Code für die binäre Suche

```
1   public static SearchResult binarySearch(int data[], int value) {
2       // search recursively
3       SearchResult res = new SearchResult();
4       if (data.length > 0) {
5           binarySearch(data, value, 0, data.length-1, res);
6       }
7       return res;
8   }
9
10  /**
11   * A recursive implementation of the binary search algorithm.
12   */
13  public static void binarySearch(int data[], int value, int left,
        int right, SearchResult res) {
14
15      // handle terminal case
16      if (left >= right) {
17          res.steps++;
18          if (data[left] == value) {
19              res.index = left;
20          }
21          return;
22      }
23
24      int mid = (left + right) / 2;
25
26      // conduct a comparison and continue recursively
27      res.steps++;
28      if (data[mid] == value) {
29          // found!
30          res.index = mid;
```

```
31          } else if (value < data[mid]) {
32              // => continue on the left side
33              binarySearch(data, value, left, mid-1, res);
34          } else {
35              // => continue on the right side
36              binarySearch(data, value, mid+1, right, res);
37          }
38      }
```

Eine mögliche rekursive Implementierung der Interpolations-Suche könnte wie folgt aussehen:

Beispiel Code für die Interpolations-Suche

```
1       public static SearchResult interpolationSearch(int data[], int
            value) {
2           // search recursively
3           SearchResult res = new SearchResult();
4           if (data.length > 0 && value >= data[0] && value <= data[data.
                length-1]) {
5               interpolationSearch(data, value, 0, data.length-1, res);
6           }
7           return res;
8       }
9
10      /**
11       * A recursive implementation of the interpolation search
                algorithm.
12       */
13      public static void interpolationSearch(int data[], int value, int
            left, int right, SearchResult res) {
14
15          // handle terminal case
16          if (left >= right) {
17              res.steps++;
18              if (data[left] == value) {
19                  res.index = left;
20              }
21              return;
22          }
23
24          int mid = Math.min(Math.max((int)(left + ((value - data[left])
                / (double)(data[right] - data[left])) * (right-left)),
                left), right);
25
26          // conduct a comparison and continue recursively
27          res.steps++;
28          if (data[mid] == value) {
29              // found!
30              res.index = mid;
31          } else if (value < data[mid]) {
32              // => continue on the left side
33              interpolationSearch(data, value, left, mid-1, res);
34          } else {
35              // => continue on the right side
36              interpolationSearch(data, value, mid+1, right, res);
37          }
38      }
```

Die beiden Lösungen unterscheiden sich abgesehen von den Methodennamen lediglich in den Zeilen 4 und 24. Zeile 4 überprüft die notwendigen Voraussetzungen für die Suche während Zeile 24 den Mittelpunkt für den nachfolgenden Schritt bestimmt.

2. Ein Best-Case Szenario für beide Such-Algorithmen ist immer dann gegeben wenn der erste Teilungspunkt den gewünschten Wert aufweist. So führt z.B. die Suche nach dem Wert 4 in dem Array [1,2,3,4,5,6,7,8] der Länge 8 innerhalb eines Schrittes in beiden Fällen zum gewünschten Ergebnis.

   Für die binäre Suche stellt die Suche nach z.B. dem ersten / letzten Element einen sehr schlechten Fall dar (genauso wie für jedes 2te Element dazwischen). Der Worst-Case tritt jedoch auch immer beim Suchen nach nicht enthaltenen Elementen auf. Die Anzahl der Schritte ist jedoch immer mit $\mathcal{O}(logn)$ begrenzt.

   Für die Interpolationssuche stellt die Suche nach dem Wert 1 in dem Array [0,0,0,0,0,0,0,8] einen Worst-Case dar. In diesem Fall werden 8 Schritte ($\mathcal{O}(n)$) benötigt.

Aufgabe 2) Heap Sort

1. Behauptung: Sei n die Höhe des Heaps. Die maximale Anzahl an Knoten innerhalb des Heaps ist mit $2^n - 1$ nach oben begrenzt.

   Beweis: mittels Induktion über die Höhe des Heaps n

   Basisfall: Der Heap ist leer, daher $n = 0$. Wir haben $2^n - 1 = 2^0 - 1 = 0$, was der Anzahl der Knoten im Baum entspricht.

   Induktionsschritt: Sei die Höhe des Heaps $n + 1$. Innerhalb des Heaps befindet sich die Wurzel sowie die beiden von ihr aus erreichbaren Teilbäume mit der maximalen Höhe n. Die Anzahl der Knoten ist daher durch $1 + 2 * (2^n - 1)$ begrenzt (unter Verwendung der Induktionshypothese). Es folgt $1 + 2 * (2^n - 1) = 1 + 2^{n+1} - 2 = 2^{n+1} - 1$ als obere Grenze für die maximale Anzahl an Knoten in einem Heap der Höhe $n + 1$.

2. Zuerst muss aus dem Array ein Heap hergestellt werden (Heap-Eigenschaft) - durch Anwenden von *downHeap* für die Positionen $n/2 - 1..0$:

   - $i = 2$: [5, 3, 17, 10, 19, 6, 22] = swap(17,22) $\Rightarrow$ [5, 3, 22, 10, 19, 6, 17]
   - $i = 1$: [5, 3, 22, 10, 19, 6, 17] = swap(3,19) $\Rightarrow$ [5, 19, 22, 10, 3, 6, 17]
   - $i = 0$: [5, 19, 22, 10, 3, 6, 17] = swap(5,22) $\Rightarrow$ [22, 19, 5, 10, 3, 6, 17]
   - $i = 0$: [22, 19, 5, 10, 3, 6, 17] = swap(5,17) $\Rightarrow$ [22, 19,17, 10, 3, 6, 5]

   In der zweiten Phase wird die Wurzel in die Resultierende Liste übertragen (hinten angereiht), durch das letzte Element im Heap ersetzt und die Heap-Eigenschaft wieder hergestellt:

   [22, 19, 17, 10, 3, 6, 5] $\Rightarrow$ [5, 19, 17, 10, 3, 6; 22] $\Rightarrow$ [19, 10, 17, 5, 3, 6; 22]
   [19, 10, 17, 5, 3, 6; 22] $\Rightarrow$ [6, 10, 17, 5, 3; 19, 22] $\Rightarrow$ [17, 10, 6, 5, 3; 19, 22]
   [17, 10, 6, 5, 3; 19, 22] $\Rightarrow$ [3, 10, 6, 5; 17, 19, 22] $\Rightarrow$ [10, 5, 6, 3; 17, 19, 22]
   [10, 5, 6, 3; 17, 19, 22] $\Rightarrow$ [3, 5, 6; 10, 17, 19, 22] $\Rightarrow$ [6, 5, 3; 10, 17, 19, 22]
   [6, 5, 3; 10, 17, 19, 22] $\Rightarrow$ [3, 5; 6, 10, 17, 19, 22] $\Rightarrow$ [5, 3; 6, 10, 17, 19, 22]
   [5, 3; 6, 10, 17, 19, 22] $\Rightarrow$ [3; 5, 6, 10, 17, 19, 22] $\Rightarrow$ [3; 5, 6, 10, 17, 19, 22]

Aufgabe 3) Beweisen von Programmen: Schleifeninvariante

1. Der Algorithmus sucht jeweils das kleinste Element innerhalb der Rest-Liste $i..length-1$ und setzt das gefundene Element ans Ende der Sortierten Teilliste $0..i-1$.

2. Schleifeninvariante: $\psi := data[pos] = min\{data[i], \ldots, data[k-1]\} \wedge k \leq data.length$

   Beweis: Annahme: $i < data.length$ (aus Kontext)

   (a) Vor Schleifeneintritt:

   $$pos = i \wedge k = i+1 \Rightarrow data[pos] = min\{data[i]\} \wedge k \leq data.length \Rightarrow \psi$$

   (b) Bei Schleifendurchlauf: Zu Beginn der Schleife gilt

   $$k < data.length \wedge data[pos] = min\{data[i], \ldots, data[k-1]\} \wedge k \leq data.length$$

   Nach Verarbeitung der If-Anweisung wird die zweite Klausel auf

   $$data[pos] = min\{data[i], \ldots, data[k-1], data[k]\}$$

   erweitert. Durch die Erhöhung von k um 1 gilt am Ende

   $$data[pos] = min\{data[i], \ldots, data[k-1]\} \wedge k \leq data.length \Leftrightarrow \psi$$

   (c) Abschluss der Schleife:

   $$\neg(k < data.length) \wedge data[pos] = min\{data[i], \ldots, data[k-1]\} \wedge k \leq data.length$$

   Aus $k < data.length \wedge k \leq data.length$ folgt $k = data.length$. Schlussendlich folgt aus

   $$k = data.length \wedge data[pos] = min\{data[i], \ldots, data[k-1]\}$$

   dass pos auf ein kleinstes Element im Bereich $i \ldots data.length - 1$ zeigt.

3. Schleifeninvariante:

   $$\begin{aligned}\psi := {} & [data[0], \ldots, data[i-1]] \ \ ist\ sortiert \wedge \\ & max\{data[0], \ldots, data[i-1]\} \leq min\{data[i], \ldots, data[data.length-1]\} \wedge \\ & i \leq data.length\end{aligned}$$

   Beweis:

   (a) Vor Schleifeneintritt:

   $$i = 0 \Rightarrow \psi$$

   da [] immer sortiert ist und $0 \leq data.length$ ebenfalls gilt.

   (b) Bei Schleifendurchlauf: Wie zuvor gezeigt ermitteln die Zeilen 4-11 das kleinste Element im Bereich $i - (data.length - 1)$, welches laut ψ größer ist als alle Element im Bereich $0 - (i-1)$. Die Zeilen 13-15 setzen das identifizierte Element an die letzte Stelle des sortierten Bereichs durch vertauschen. Hierdurch wird der sortierte Bereich um eine Stelle erweitert. Durch das Inkrementieren von i werden die Indices so angepasst, dass am Ende der Schleife ψ erneut erfüllt ist.

   (c) Abschluss der Schleife: Aus

   $$\neg(i < data.length) \wedge \psi \Rightarrow \psi \wedge i = data.length$$

   woraus wiederum folgt, dass der gesamte Bereich $[data[0], \ldots, data[data.length-1]]$ sortiert ist.

Aufgabe 4) Teilmengen In der Praxis würde man dieses Beispiel vermutlich über eine Hashtabelle realisieren ($\mathcal{O}(1)$ für einfügen / suchen), wodurch alle Operationen in $\mathcal{O}(max(|A|, |B|))$ realisiert werden können. Diese Datenstruktur wird später in der Vorlesung behandelt.

Jedoch lassen sich bereits recht brauchbare Lösungen mit den bekannten Such/Sortier-Algorithmen zusammenstellen:

1. Um $A \subseteq B$ zu überprüfen, kann man z.B. B sortieren ($\mathcal{O}(nlogn)$, wobei $n = |B|$. Im Anschluss werden sämtliche Elemente von A im sortierten B gesucht (e.g. binäre Suche). Hierdurch erhält man eine Gesamtkomplexität von $\mathcal{O}(n_B log n_B) + n_A * \mathcal{O}(log n_B)$. Da $n_A \leq n_B$ kann die Komplexität somit mit $\mathcal{O}(n_B log n_B)$ angegeben werden.
2. Um $A = B$ zu testen kann $A \subseteq B \wedge |A| = |B|$ überprüft werden. Die Komplexität steigt dadurch nicht. Beide Arrays zu sortieren und zu vergleichen liegt in derselben Komplexitätsklasse, weist jedoch eine höhere tatsächliche Laufzeit auf.