Halden

Weiterführende Literatur:

- *Algorithmen und Datenstrukturen: Tafelübung 11, WS 2018/19*, Seite 25-32
- *Wikipedia-Artikel „Heap (Datenstruktur)“*
- Saake und Sattler, *Algorithmen und Datenstrukturen*, Seite 407-409 Kapitel 14.6.1

Visualisierungstools

- Min-Heap: `https://www.cs.usfca.edu/~galles/visualization/Heap.html`
- Max-Heap: `https://visualgo.net/en/heap`
- Max-Heap: `http://btv.melezinek.cz/binary-heap.html`

Ein Heap (englisch wörtlich: *Haufen* oder *Halde*) ist eine auf Bäumen basierende *abstrakte* und *dynamische* Datenstruktur. Der Begriff Heap wird häufig als bedeutungsgleich zu einem *partiell geordneten Binärbaum* verstanden.[1]

Haufen
Halde
abstrakte
dynamische
partiell geordneten Binärbaum

Die Datenstruktur Heap bezeichnet einen binären Baum, der folgende Eigenschaften erfüllt:[2]

- Der Baum ist vollständig, d. h. , die Blattebene ist von links nach rechts gefüllt.
- Der Schlüssel eines jeden Knotens ist kleiner (oder gleich) als die Schlüssel seiner Kinder. Diese partielle Ordnung wird auch als Heap-Eigenschaft bezeichnet.

Es gibt zwei unterschiedlichen Arten von Halden, nämlich:

- ein *Max-Heap*. Hier ist die Wurzel jedes Teilbaums ist das *Maximum* aller Knoten des Teilbaums.
- eine *Min-Heap*: Die Wurzel jedes Teilbaums ist das *Minimum* aller Knoten des Teilbaums.

Max-Heap
Maximum
Min-Heap
Minimum

Duplikate sind in einem Heap *erlaubt*. Heaps erlauben einen schnellen Zugriff auf das *größte* bzw. *kleinste* Element. Sie werden deshalb als *Prioritätswarteschlange* verwendent. Heaps können als „klassischer“ Binärbaum implementiert werden. Eine effiziente Speicherung ist in einem *Array* möglich.[3]

Duplikate
erlaubt
größte
kleinste
Prioritätswarteschlange
Array

Links-Vollständigkeit

Oft (nicht immer) verstehen wir unter Heaps links-vollständige Bäume: d. h. alle „Ebenen“ (bis auf die unterste) sind *voll besetzt* und auf unterster „Ebene“ sitzen alle Knoten *soweit links wie möglich*. Dadurch ist eine *lückenlose Darstellung* in einem *Array* möglich (sog. Feld-Einbettung). Nicht nur Halden können

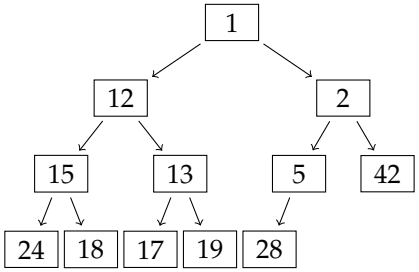
voll besetzt
soweit links wie möglich
lückenlose Darstellung

linksvollständig sein, sondern jeder beliebige Baum.

Array

Berechnung der Indizes

- *Wurzel* steht an Position 0
- *Kinder* von Knoten an Position i stehen an Stelle $2 \cdot i + 1$ und $2 \cdot i + 2$
- *Elternknoten* von Knoten an Position i steht an Stelle $\frac{i-1}{2}$[4]



Operationen

Einfügen eines neuen Elements

nächste freie Position in der untersten Ebene einfügt

voll

erster Knoten einer neuen Ebene

nach oben wandern

Ein neues Element wird an die *nächste freie Position in der untersten Ebene einfügt*. Falls die Ebene *voll* ist, wird das neue Element *erster Knoten einer neuen Ebene*. Solange die Halden-Eigenschaft in einem Teilbaum verletzt ist, lassen wir das neue Element entsprechend der Halden-Eigenschaft *nach oben wandern*.

Löschen eines Elements

ersetzen

letzten Element der untersten Ebene

nach unten sickern

Min-Heap

kleineren Kind

Max-Heap

größeren Kind

Wir *ersetzen* das zu löschende Element mit dem *letzten Element der untersten Ebene*. Solange die Halden-Eigenschaft in einem Teilbaum verletzt ist, lassen wir das neue Element entsprechend der Halden-Eigenschaft *nach unten sickern*. Bei einer *Min-Heap* wird mit dem *kleineren Kind* getauscht, bei einer *Max-Heap* mit dem *größeren Kind*.[5]

```
3  import java.util.Arrays;
4
5  import org.bschlangaul.baum.visualisierung.BaumReporter;
6  import org.bschlangaul.baum.visualisierung.StummerBaumReporter;
7
8  enum HaldenTyp {
9    MIN, MAX
10 }
11
12 @SuppressWarnings("unchecked")
13
```

[1] *Wikipedia-Artikel „Heap (Datenstruktur)“.*
[2] Saake und Sattler, *Algorithmen und Datenstrukturen*, Seite 407, Kapitel 14.6.1.
[3] *Algorithmen und Datenstrukturen: Tafelübung 11, WS 2018/19*, Seite 22.
[4] *Algorithmen und Datenstrukturen: Tafelübung 11, WS 2018/19*, Seite 26.
[5] *Algorithmen und Datenstrukturen: Tafelübung 11, WS 2018/19*, Seite 28.

```java
/**
 * Feld-Implementation einer Halde (nach
 * <a href="https://gist.github.com/snarkbait/86c7a4bc743e8f327dbc">Philboyd
 * Studge</a>
 */
public class Halde<T extends Comparable<T>> {

  public BaumReporter reporter = new StummerBaumReporter();

  private static final int STANDARD_KAPAZITÄT = 10;

  private T[] halde;
  private int füllstand;

  /**
   * Der aktuelle Füllstand der Halde. Er wird hochgezählt, wenn ein neuer
   * Schlüsselwert eingefügt wird und er wird erniedrigt, wenn ein Schlüsselwert
   * entnommen wird.
   */
  private HaldenTyp typ;

  /**
   * Der Standard-Konstruktor.
   *
   * @param typ Ob es sich um eine Min-Halde oder eine Max-Halde handeln soll.
   */
  public Halde(HaldenTyp typ) {
    halde = (T[]) new Comparable[STANDARD_KAPAZITÄT];
    füllstand = 0;
    this.typ = typ;
  }

  /**
   * Gib das Feld (Array) zurück, das die Schlüsselwerte der Halde speichert.
   *
   * @return Das Feld (Array) mit den Schlüsselwerten.
   */
  public T[] gibHaldenFeld() {
    return Arrays.copyOfRange(halde, 1, füllstand + 1);
  }

  /**
   * adds a generic type T to heap
   * <p>
   * percolates down the tree
   *
   * @param value type T value
   */
  public void fügeEin(T value) {
    reporter.berichteÜberschrift("Nach dem Einfügen von „" + value + """, 0);
    if (this.füllstand >= halde.length - 1) {
      halde = this.vergrößern();
    }
    füllstand++;
    halde[füllstand] = value;
    steigeAuf();
    berichteBaum(0);
  }

  /**
   * Removes min or max item from heap
   * <p>
```

```
76      * re-heapifies
77      *
78      * @return value of T that is minimum or maximum value in heap
79      */
80     public T entferne() {
81       T result = peek();
82       vertausche(1, füllstand);
83       halde[füllstand] = null;
84       füllstand--;
85       versickere();
86       return result;
87     }
88
89     /**
90      * Remove specific object from heap
91      *
92      * @param value type T
93      * @return true if found and removed
94      */
95     public boolean entferne(T value) {
96       for (int i = 0; i < halde.length; i++) {
97         if (value.equals(halde[i])) {
98           reporter.berichteÜberschrift("Nach dem Entfernen von „" + value + """,
              ↪  0);
99           vertausche(i, füllstand);
100          halde[füllstand] = null;
101          füllstand--;
102          versickere();
103          berichteBaum(0);
104          return true;
105        }
106      }
107      return false;
108    }
109
110    /**
111     * Removes min or max item from heap same as <code>remove()</code> but does not
112     * throw exception on empty
113     * <p>
114     * re-heapifies
115     *
116     * @return value of T that is minimum or maximum value in heap; or
117     *         <code>null</code> if empty
118     */
119    public T poll() {
120      if (istLeer())
121        return null;
122      T result = peek();
123      vertausche(1, füllstand);
124      halde[füllstand] = null;
125      füllstand--;
126      versickere();
127      return result;
128    }
129
130    /**
131     * Checks if heap is empty
132     *
133     * @return <code>true</code> if empty
134     */
135    public boolean istLeer() {
136      return füllstand == 0;
```

```
137     }
138
139     /**
140      * returns min/max value without removing it
141      *
142      * @return value type T
143      * @throws IllegalStateException if empty
144      */
145     public T peek() {
146       if (istLeer())
147         throw new IllegalStateException();
148       return halde[1];
149     }
150
151     /**
152      * Length/size of heap
153      *
154      * @return int size of heap
155      */
156     public int length() {
157       return füllstand;
158     }
159
160     /**
161      * Add DEFAULT_CAPACITY to length of <code>heap</code> array
162      *
163      * @return new array of type T
164      */
165     private T[] vergrößern() {
166       return Arrays.copyOf(halde, halde.length + STANDARD_KAPAZITÄT);
167     }
168
169     /**
170      * Die Methode wird oft „bubbleUp" genannt.
171      */
172     private void steigeAuf() {
173       int index = füllstand;
174       if (typ == HaldenTyp.MIN) {
175         while (hatEltern(index) &&
            ↪  (gibElternSchlüssel(index).compareTo(halde[index]) > 0)) {
176           vertausche(index, gibIndexEltern(index));
177           index = gibIndexEltern(index);
178         }
179       } else {
180         while (hatEltern(index) &&
            ↪  (gibElternSchlüssel(index).compareTo(halde[index]) < 0)) {
181           vertausche(index, gibIndexEltern(index));
182           index = gibIndexEltern(index);
183         }
184       }
185     }
186
187     /**
188      * Die Methode wird oft „percolate" oder „bubbleDown" genannt.
189      */
190     private void versickere() {
191       int index = 1;
192       if (typ == HaldenTyp.MIN) {
193         while (hatLinks(index)) {
194           int kleiner = gibIndexLinks(index);
195           if (hatRechts(index) &&
              ↪  halde[gibIndexLinks(index)].compareTo(halde[gibIndexRechts(index)]) >
              ↪  0) {
```

```
196             kleiner = gibIndexRechts(index);
197           }
198           if (halde[index].compareTo(halde[kleiner]) > 0) {
199             vertausche(index, kleiner);
200           } else
201             break;
202           index = kleiner;
203         }
204       } else {
205         while (hatLinks(index)) {
206           int größer = gibIndexLinks(index);
207           if (hatRechts(index) &&
              ↪  halde[gibIndexLinks(index)].compareTo(halde[gibIndexRechts(index)]) <
              ↪  0) {
208             größer = gibIndexRechts(index);
209           }
210           if (halde[index].compareTo(halde[größer]) < 0) {
211             vertausche(index, größer);
212           } else
213             break;
214           index = größer;
215         }
216       }
217     }
218
219     private T gibSchlüssel(int index) {
220       return halde[index];
221     }
222
223     /**
224      * Überprüfe ob der Knoten einen Elternknoten hat.
225      *
226      * @param index Die Index-Nummer des Knoten.
227      *
228      * @return Wahr wenn der Knoten einen Elternknoten hat.
229      */
230     private boolean hatEltern(int index) {
231       return index > 1;
232     }
233
234     /**
235      * Berechne die Index-Nummer des linken Kindknotens.
236      *
237      * @param index Die Index-Nummer des Knoten.
238      *
239      * @return Die Index-Nummer des linken Kindknotens.
240      */
241     private int gibIndexLinks(int index) {
242       return index * 2;
243     }
244
245     /**
246      * Berechne die Index-Nummer des rechten Kindknotens.
247      *
248      * @param index Die Index-Nummer des Knoten.
249      *
250      * @return Die Index-Nummer des rechten Kindknotens.
251      */
252     private int gibIndexRechts(int index) {
253       return index * 2 + 1;
254     }
255
```

```
256     /**
257      * Berechne die Index-Nummer des Elternknoten.
258      *
259      * @param index Die Index-Nummer des Knoten.
260      *
261      * @return Die Index-Nummer des Elternknoten.
262      */
263     private int gibIndexEltern(int index) {
264       return index / 2;
265     }
266
267     /**
268      * Überprüfe, ob der Knoten einen linken Kindknoten hat.
269      *
270      * @param index Die Index-Nummer des Knoten.
271      *
272      * @return Wahr, wenn der Knoten ein linkes Kind hat.
273      */
274     private boolean hatLinks(int index) {
275       return gibIndexLinks(index) <= füllstand;
276     }
277
278     /**
279      * Überprüfe, ob der Knoten einen rechten Kindknoten hat.
280      *
281      * @param index Die Index-Nummer des Knoten.
282      *
283      * @return Wahr, wenn der Knoten ein rechtes Kind hat.
284      */
285     private boolean hatRechts(int index) {
286       return gibIndexRechts(index) <= füllstand;
287     }
288
289
290     /**
291      * get parent value
292      *
293      * @param index Die Index-Nummer des Knoten.
294      *
295      * @return Der Schlüsselwert des Elternknoten.
296      */
297     private T gibElternSchlüssel(int index) {
298       return halde[gibIndexEltern(index)];
299     }
300
301     /**
302      * Vertausche die Schlüsselwerte im Feld {@link halde}.
303      *
304      * @param index1 Die Indexnummer des ersten Schlüsselwertes, der vertauscht
305      *               werden soll.
306      * @param index2 Die Indexnummer des zweiten Schlüsselwertes, der vertauscht
307      *               werden soll.
308      */
309     private void vertausche(int index1, int index2) {
310       T schlüssel1 = gibSchlüssel(index1);
311       T schlüssel2 = gibSchlüssel(index2);
312       berichteBaum(String.format("Nach Vertauschen von „%s" und „%s"", schlüssel1,
          ↪  schlüssel2), 1);
313
314       T tmp = halde[index1];
315       halde[index1] = halde[index2];
316       halde[index2] = tmp;
```

```
317     }
318
319     /**
320      * Konvertiere die Halde zu Text.
321      *
322      * @return Alle Werte der Halde als Text mit Kommas zusammengehängt.
323      */
324     @Override
325     public String toString() {
326       String ausgabe = "";
327       for (T schlüssel : gibHaldenFeld()) {
328         ausgabe += schlüssel + ", ";
329       }
330       return ausgabe;
331     }
332
333     /**
334      * Exportiere die Halde als Binärbaum. Hier kann kein „normaler" binärer
335      * Suchbaum verwendet werden, da in diesem Baum ganz andere Einfügeregeln
336      * gelten. Deshalb schummeln wir hier einen Baum, damit wir ihn darstellen
337      * können.
338      *
339      * @return Ein Repräsentation als Binärbaum
340      */
341     public BinaerBaum gibBinaerBaum() {
342       BinaererSuchBaum baum = new BinaererSuchBaum();
343
344       T[] haldenFeld = gibHaldenFeld();
345
346       berichteHaldenFeldAlsTabelle(haldenFeld);
347
348       BaumKnoten[] knoten = new BaumKnoten[haldenFeld.length];
349       for (int i = 0; i < haldenFeld.length; i++) {
350         knoten[i] = new BaumKnoten(haldenFeld[i]);
351       }
352
353       for (int i = 0; i < haldenFeld.length; i++) {
354         BaumKnoten k = knoten[i];
355         // Die Halde setzt den ersten Wert auf das 2. Element des Feldes.
356         if (hatLinks(i + 1))
357           k.setzeLinks(knoten[gibIndexLinks(i + 1) - 1]);
358         if (hatRechts(i + 1))
359           k.setzeRechts(knoten[gibIndexRechts(i + 1) - 1]);
360       }
361       // Der erste Knoten ist auf rechts gesetzt.
362       baum.kopf.setzeRechts(knoten[0]);
363       return baum;
364     }
365
366     private void berichteHaldenFeldAlsTabelle(T[] haldenFeld) {
367       String[] kopfZeile = new String[haldenFeld.length];
368       for (int i = 0; i < haldenFeld.length; i++) {
369         kopfZeile[i] = String.valueOf(i);
370       }
371
372       String[][] zeilen = new String[1][haldenFeld.length];
373
374       for (int i = 0; i < haldenFeld.length; i++) {
375         zeilen[0][i] = String.valueOf(haldenFeld[i]);
376       }
377
378       reporter.berichteTabelle(kopfZeile, zeilen);
```

```
379     }
380
381     public void berichteBaum(int redselig) {
382       BinaerBaum haldenBaum = gibBinaerBaum();
383       reporter.berichteBaum(haldenBaum, redselig);
384     }
385
386     public void berichteBaum(String überschrift, int redselig) {
387       BinaerBaum haldenBaum = gibBinaerBaum();
388       reporter.berichteBaum(haldenBaum, überschrift, redselig);
389     }
390
391   }
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/baum/Halde.java`

```
3   /**
4    * Sortiere ein Zahlen-Feld mit Hilfe des Heapsort-Algorithmus. (Nach Saake
5    * Seite 412)
6    */
7   public class Heap extends Sortieralgorithmus {
8     private void versickere(int index, int letzterIndex) {
9       int i = index + 1, j;
10      // zahlen[i] hat linken Sohn
11      while (2 * i <= letzterIndex) {
12        // zahlen[j] ist linker Sohn von zahlen[i]
13        j = 2 * i;
14        // zahlen[i] hat auch rechten Sohn
15        if (j < letzterIndex) {
16          if (zahlen[j - 1] < zahlen[j]) {
17            // zahlen[j] ist jetzt kleiner
18            j++;
19          }
20        }
21
22        if (zahlen[i - 1] < zahlen[j - 1]) {
23          vertausche(i - 1, j - 1);
24          // versickere weiter
25          i = j;
26        } else {
27          // halte an, heap-Bedingung erfüllt
28          break;
29        }
30      }
31    }
32
33    /**
34     * Sortiere ein Zahlen-Feld mit Hilfe des Heapsort-Algorithmus.
35     *
36     * @return Das sortierte Zahlenfeld.
37     */
38    public int[] sortiere() {
39      int i;
40      for (i = zahlen.length / 2; i >= 0; i--) {
41        versickere(i, zahlen.length);
42      }
43
44      for (i = zahlen.length - 1; i > 0; i--) {
45        // tauscht jeweils letztes Element des Heaps mit dem ersten
46        vertausche(0, i);
47        // heap wird von Position 0 bis i hergestellt
48        versickere(0, i);
49      }
```

```
50      return zahlen;
51    }
52
53    public static void main(String[] args) {
54      new Heap().teste();
55    }
56  }
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/sortier/Heap.java`

Literatur

[1] *Algorithmen und Datenstrukturen: Tafelübung 11, WS 2018/19*. `https://www.studon.fau.de/file2567217_download.html`. FAU: Lehrstuhl für Informatik 2 (Programmiersysteme).

[2] Gunter Saake und Kai-Uwe Sattler. *Algorithmen und Datenstrukturen. Eine Einführung in Java*. 2014.

[3] *Wikipedia-Artikel „Heap (Datenstruktur)“*. `https://de.wikipedia.org/wiki/Heap_(Datenstruktur)`.