1 Heap-Sort

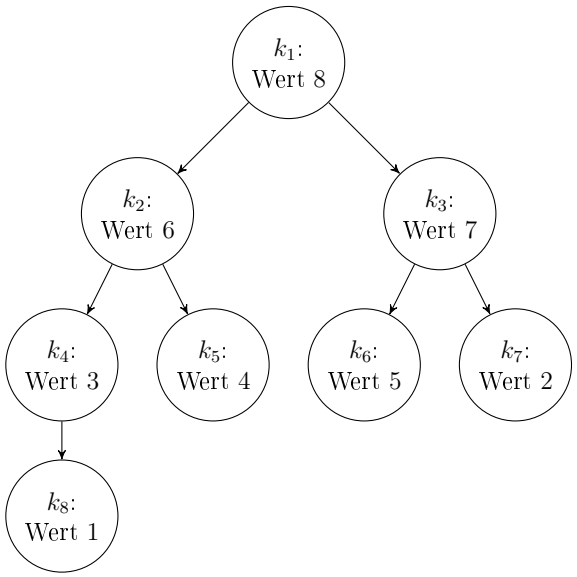
Während der Vorlesung wurde eine Verfahren eingeführt, um Zahlenreihen der Länge N zu sortieren, das sogenannte Sortieren durch Einfügen. Wir hatten auch gesehen, dass dieses Verfahren im schlechtesten Fall $\mathcal{O}(N^2)$ Vergleichs- und Vertauschungs-Operationen benötigt. Wir werden nun ein Verfahren einführen, dass im schlechtesten Fall $\mathcal{O}(N \log(N))$ Vergleichs- und Vertauschungs-Operationen benötigt. Es wird *Heap-Sort* genannt, weil es auf einer Datenstruktur basiert, die man Halde oder *Heap* nennt.

Definieren wir zunächst, was man einen Heap nennt. Eine Folge von Schlüsseln $F = k_1, k_2, \ldots, k_N$ nennt man einen Heap, wenn

$$k_i \;\leq\; k_{\lfloor i/2 \rfloor}\,, \qquad 1 \leq i \leq N\,.$$

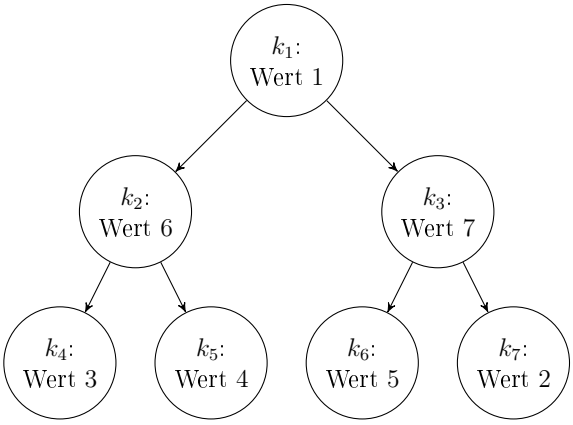
Anders ausgedrückt bedeutet dies $k_i \geq k_{2i}$ und $k_i \geq k_{2i+1}$, sofern $2i \leq N$ bzw. $2i+1 \leq N$. Die Schlüssel k_{2i} und k_{2i+1} nennt man Nachfolger von k_i. Wichtig ist, dass zwar $k_i \geq k_{2i}$ und $k_i \geq k_{2i+1}$, aber der Heap keine Relation zwischen k_{2i} und k_{2i+1} impliziert. Per Definition eines Heaps ist natürlich $k_1 \geq k_i$ für alle $1 < i \leq N$.

Unter den Schlüsseln $k_1, \ldots, k_N$ können Sie sich zunächst einfach ganze Zahlen vorstellen. Aber im Prinzip können es allgemeine Daten sein, für die die Operation $\leq$ definiert ist. Beispielsweise ist $F = 8, 6, 7, 3, 4, 5, 2, 1$ ein Heap mit $N = 8$ im Sinne obiger Definition, wie man leicht verifiziert. Man stellt einen Heap am einfachsten mit Hilfe eines sogenannten balancierten, binären Baumes dar. Man nennt einen solchen binären Baum balanciert, wenn seine Höhe minimal ist für die Anzahl N an Elementen. Für das Beispiel sieht der binäre Baum wie folgt aus:

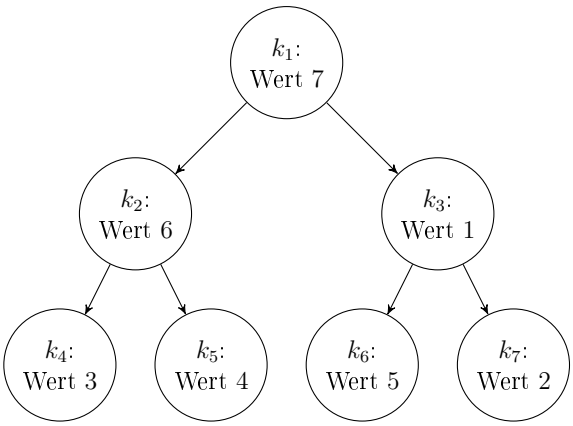


Wir werden für balancierte, binäre Bäume der Konvention folgen, dass immer von links aufgefüllt wird. Falls also ein Vertex keinen linken Nachfolger hat, so hat der Vertex auch keinen rechten Nachfolger.

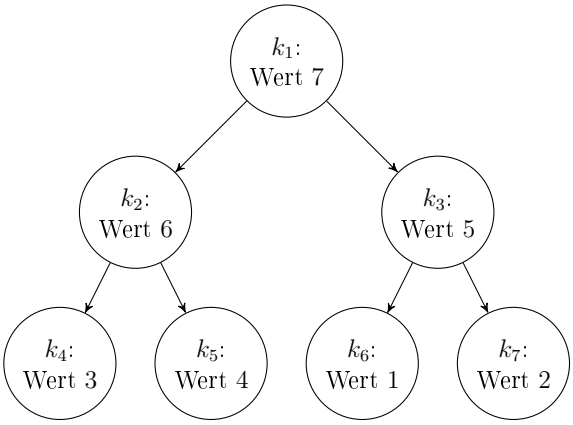
Man sieht in diesem Beispiel auch sofort, dass jeder Unterbaum eines Heaps auch selbst wieder ein Heap ist. In unserem Beispiel bleiben beispielsweise nach Entfernen des ersten Elements zwei Teilheaps zurück, als $F_1 = 6, 3, 4, 1$ und $F_2 = 7, 5, 2$. Zum Sortieren entfernen wir das erste Element, also k_1 aus dem Heap. Dann ersetzen wir es durch das Element mit dem größten Index, im Beispiel also k_8. Damit erhalten wir einen neuen Binärbaum, der allerdings die Heap Eigenschaften nicht erfüllt, denn 1 ist nicht größer als 6 und 7.



Der neue Binärbaum wird wieder zu einem Heap, indem man den neuen Schlüssel k_1 *versickern* lassen. Dies geschieht, indem wir ihn immer wieder mit dem größeren seiner beiden Nachfolger vertauschen. Im Beispiel erhält man also nach dem ersten Versickerungsschritt



und nach dem zweiten



nach dem die Heapeigenschaft wieder hergestellt ist. Diese Schritte wiederholt man, bis der Heap keine Schlüssel mehr enthält. Für ganze Zahlen könnte man das Versickern mit Hilfe eines Arrays wie folgt realisieren:

```
1:  procedure VERSICKERN(a, i, m)
2:      input array a, integer i, m
3:      output heap a
4:      while 2i + 1 ≤ m do                    ▷ a[i] hat linken Nachfolger
5:          j = 2i + 1                          ▷ a[j] ist linker Nachfolger
6:          if j < m then
7:              if a[j] < a[j + 1] then
8:                  j ← j + 1                   ▷ a[j] ist jetzt groesster Nachfolger
9:              end if
10:         end if
11:         if a[i] < a[j] then                 ▷ Vertausche mit groesserem Nachfolger
12:             a[j] ↔ a[i]
13:             i = j
14:         else
15:             i = m                           ▷ Heapbedingung wieder erfuellt
16:         end if
17:     end while
18: end procedure
```

Hierbei ist m die Länge des Arrays und i der Index des Elements, vom dem das Versickern starten soll. Sobald wir also eine Folge mit Heapbedingung haben, können wir durch wiederholtes Anwenden dieses Verfahrens die Folge sortieren. Allerdings müssen wir die Folge dafür erstmal in einen Heap verwandeln. Dies kann man erreichen, indem man die Schlüssel $k_{\lfloor N/2 \rfloor}$ bis k_1 versickern. Dadurch werden schrittweise immer größere Heaps aufgebaut, bis am

Ende schließlich der Ausgangsheap vorliegt. Eine mögliche Implementierung könnte wie folgt aussehen:

1: **procedure** HEAPSORT(a, N)
2: **Input** a, N
3: **Output** a
4: **for** $i = N/2-1, N/2-2, \ldots, 0$ **do** ▷ Erzeuge initialen Heap
5: VERSICKERN($a, i, N-1$)
6: **end for**
7: **for** $i = N-1, N-2, \ldots, 1$ **do** ▷ Sortieren durch Versickern
8: $a[i] \leftrightarrow a[0]$
9: VERSICKERN($a, 0, i-1$)
10: **end for**
11: **end procedure**

Implementieren Sie den Heap-Sort Algorithmus für eine Folge reeller Zahlen variabler Länge.

Bemerkung: Das Verfahren Heap-Sort sortiert eine Folge im schlechtesten Fall in $\mathcal{O}(N \log N)$ Schritten. Das $\log N$ kommt aus der Höhe des Baumes.

Wir werden nun anstelle des Arrays eine spezielle Art sogenannter verketteter Listen verwenden. Der Vorteil dieser verketteten Liste ist, dass keine Daten mehr kopiert werden müssen, sondern lediglich Zeiger auf solche Daten. Das kann man zwar für Heapsort auch mit Indexarrays erreichen, aber als Übung betrachten wir nun verkettete Listen. Betrachtet man einen Vertex in einem binären Baum, so ist er gekennzeichnet durch einen oder keinen Vorgängervertex, keinen, einen oder zwei Nachfolgervertizes und ein Datum. Folgender Datentyp

```
typedef struct element
{
  Datum datum;              // die eigentlichen Daten
  struct element *left;     // Zeiger auf linken Nachfolger
  struct element *right;    // Zeiger auf rechten Nachfolger
  struct element *parent;   // Zeiger auf den Vorgaenger
} element;
```

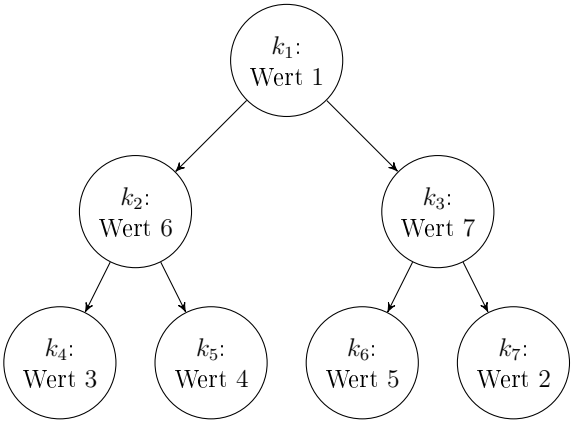
implementiert diese Eigenschaften. Die eigentlichen Daten sind in `datum` vom Typ `Datum` gespeichert. Wir werden den `NULL`-Zeiger verwenden, um keinen Vorgänger oder Nachfolger zu kennzeichnen. Dementsprechend wird für die Wurzel des Baumes `parent = NULL;` gesetzt. Blätter des Baumes mit nur einem oder keinem Nachfolger haben `left = NULL` und/oder `right = NULL`. `Datum` kann ein beliebiger Datentyp sein, für den eine Vergleichsoperation $\leq$ definiert ist.

Im Prinzip können wir nun oben vorgestellten Algorithmus auf diesen Datentyp umstellen.

Dafür brauchen wir zunächst eine Funktion, die den Vergleich durchführt. In Pseudo-Code wäre das das folgende

```
1: procedure COMPARE(e1, e2)
2:     input Elemente e1, e2 vom Typ element
3:     output 0 oder 1
4:     if e1.datum ≤ e2.datum then
5:         return 1
6:     else
7:         return 0
8:     end if
9: end procedure
```

Für das Versickern lassen brauchen wir außer der Vergleichsoperation noch das Vertauschen von zwei Elementen. Betrachten wir noch einmal folgenden Baum, der die Heapeigenschaft nicht erfüllt



Um k_1 mit k_3 zu vertauschen, muss der linke Nachfolger von k_3 auf k_2, der rechte auf k_1 und der Vorgänger auf `NULL` gesetzt werden. Genauso muss der Vorgänger von k_1 auf k_3, und der linke bzw. rechte Nachfolger auf k_6 bzw. k_7. Im Pseudo-Code sähe das wie folgt aus

```
1: procedure SWAP(parent, child)
2:     InOutput: parent, child vom Typ element
3:     child.parent = parent.parent
4:     parent.parent = child
5:     if child == parent.left then
6:         parent.left = child.left
7:         child.left = parent
8:         child.right ↔ parent.right
```

```
    else
        parent.right = child.right
        child.right = parent
        child.left ↔ parent.left
    end if
end procedure
```

Hierbei ist es wichtig, das bekannt ist, welches Element der Vorgänger und welches der Nachfolger ist. Man beachte, dass all diese Operationen lediglich Operationen auf Zeigern sind. Es werden also keine Daten kopiert, wenn man die Parameter `parent` und `child` als Zeiger übergibt. Das gilt für alle nun folgenden Funktionen: Sie sollten *call-by-reference* verwenden. Damit kann man das Versickern implementieren. Erstellen Sie zunächst Pseudo-Code für das Versickern mit dem Datentyp `element`. Eine solche VERSICKERN-Funktion bekommt nur noch das Wurzelelement des (Unter-)Baumes übergeben, keine weiteren Parameter.

Etwas schwieriger als mit Arrays ist es, den anfänglichen Heap zu erzeugen. Zunächst erzeugt man dafür einen balancierten binären Baum mit N Elementen. Nehmen wir an, dass wir einen solchen Baum erzeugt haben und das Wurzelelement dieses zufälligen, balancierten binären Baumes das Element `root` ist. Diesen binären Baum kann man zu einem Heap machen, indem man rekursiv die Funktion VERSICKERN anwendet. Das Vorgehen ist dabei sehr ähnlich wie beim Heapsort mit Arrays. Man definiert folgende Funktion

```
procedure HEAPIFY(root)
    Input root
    if root.left==NULL then
        return                                  ▷ root hat keine Nachfolger
    else
        HEAPIFY(root.left)                      ▷ root hat einen linken Nachfolger
    end if
    if root.right! =NULL then
        HEAPIFY(root.right)                     ▷ root hat einen rechten Nachfolger
    end if
    VERSICKERN(root)
end procedure
```

Wie man sieht, handelt es sich um eine rekursive Funktion. Sie ruft sich selbst wieder auf, bis ein Vertex ohne Nachfolger erreicht wird. Wird die Funktion HEAPIFY mit dem Wurzelelement `root` des zufälligen, balancierten binären Baumes aufgerufen, so wird HEAPIFY so lange wieder aufgerufen, bis kein linker Nachfolger mehr existiert. Dann wird eine Ebene darüber VERSICHERN aufgerufen. So wird sukzessive von den Blättern des Baumes an für jede Ebene mit der Funktion VERSICKERN die Heapeigenschaft hergestellt. Man nennt dieses Verfahren

auch Tiefensuche. Implementieren Sie die Funktion VERSICKERN und die Funktion HEAPIFY in `C`.

Zurück zum Erzeugen des zufälligen, balancierten binären Baumes. Dabei können Sie wie folgt vorgehen: Starten Sie mit einem Zeigerarray

```
1  element * a[];
```

Reservieren Sie Speicher für N Elemente vom Typ `element*` und lassen Sie die Elemente von $a[]$ entsprechend auf ihre Daten zeigen. Nun können Sie rekursiv (genau wie in der Funktion HEAPIFY) einen zufälligen, balancierten binären Baum erzeugen. Machen Sie sich dafür klar, dass auf der p-ten Ebene des Baumes maximal 2^p Vertizes existieren können, wobei die Ebene des Wurzelelements $p = 0$ hat. Außerdem muss eine Ebene erst vollständig gefüllt sein, bevor eine nächste Ebene begonnen werden darf. Darüberhinaus rufen Sie sich noch einmal ins Gedächtnis, dass in der Array-Implementierung von oben die Indizes der Nachfolger von Element i durch $2i$ und $2i + 1$ gegeben sind. Das Array $a[]$ können Sie für das Speichern der sortierten Folge wieder verwenden.

Nun fehlen nur noch Kleinigkeiten für den kompletten Heapsort Algorithmus:

1. Wir benötigen eine Funktion FINDLAST, die das letzte Element im Baum findet. Dabei sollte man verwenden, dass der Baum balanciert ist.

2. Weiterhin brauchen wir eine Funktion REMOVE, die ein Element aus dem Baum entfernt. Es dürfen dabei nur Elemente entfernt werden, die keine Nachfolger haben.

   Sinnvollerweise kombiniert man FINDLAST und REMOVE. Man schreibt also REMOVE so, dass nur Elemente entfernt werden dürfen, wenn das Element keine Nachfolger hat und nach dem Entfernen der binäre Baum noch balanciert ist.

3. Und wir benötigen eine Funktion REPLACEROOT, die das Wurzelelement durch ein anderes Element ersetzt.

Diese drei, bzw. zwei Funktionen sollten ohne große Schwierigkeiten implementierbar sein. Entwerfen Sie Pseudo-Code für die Funktionen und implementieren Sie sie anschließend in `C`.