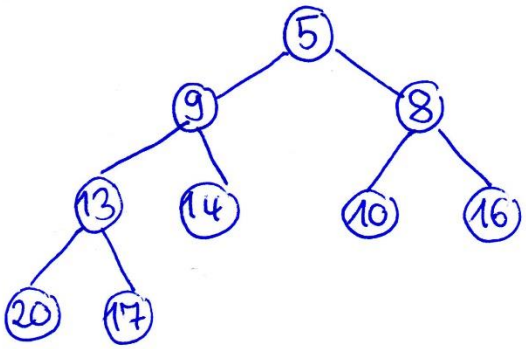
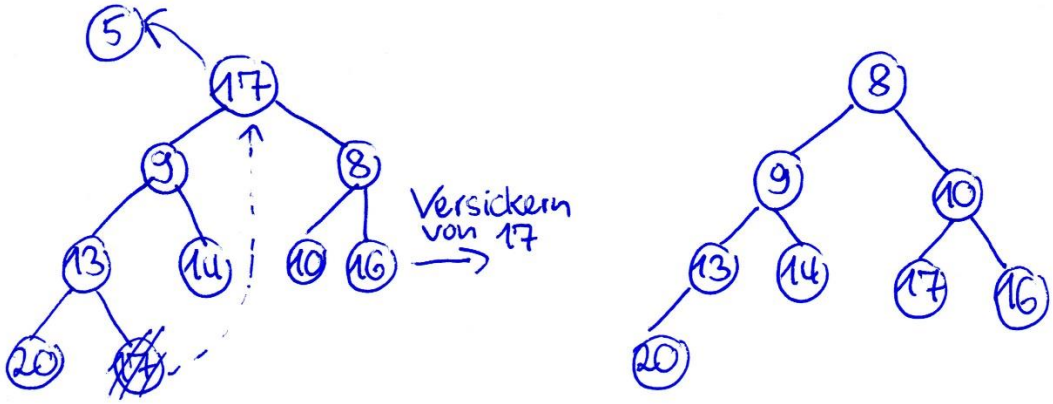
Lösung 12: Priority Queue, Substring Search

Aufgabe 1: Implementierung der ADT Priority Queue über einen MinHeap

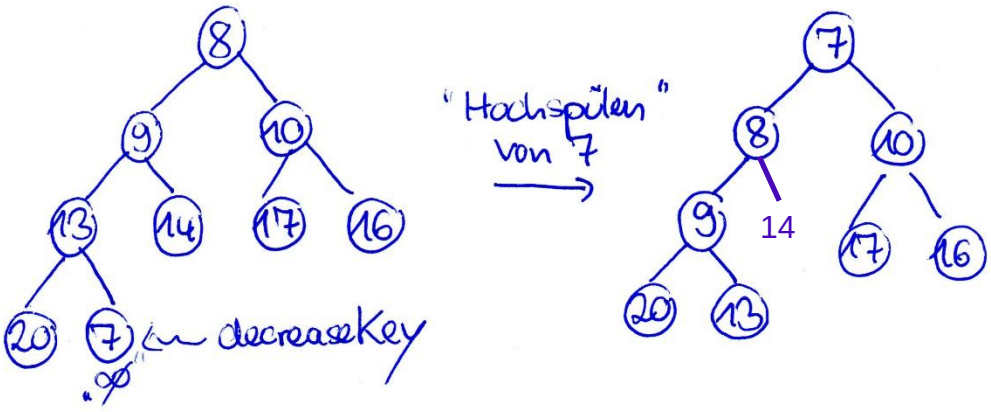
a) Die Heap-Eigenschaft ist bereits erfüllt, keine Anpassungen notwendig!



b) Zunächst wird die Wurzel (5) entfernt, anschließend wird das Element 17 von ganz unten rechts im Heap (größter Arrayindex) an die Wurzel gebracht. Nun muss die Wurzel „versickert“ werden (MIN-HEAPIFY).



c) Das Element wird zunächst mit dem Wert „unendlich“ ganz rechts unten eingefügt. Anschließend wird DECREASE-KEY aufgerufen, wodurch das Element durch schrittweises Vertauschen mit dem Elternknoten nach oben „gespült“ wird.



d)

- `MINIMUM`: O(1), das Minimum befindet sich an der Wurzel.
- `EXTRACT-MIN`: O(log n), siehe Vorlesung.
- `INSERT`: O(log n), siehe Vorlesung.

e)

- `MINIMUM`: O(log n), das Minimum ist das Element ganz links unten im Baum.
- `EXTRACT-MIN`: O(log n): Man muss erst das Minimum suchen (O(log n)). Beim Entfernen des Minimums können dann zusätzlich die Bedingungen an einen Rot-Schwarz-Baum verletzt werden. Deshalb muss der Red-Black-Tree ggfs. repariert werden (zusätzlich: O(log n)).
- `INSERT`: O(log n), Suchen der Einfügeposition + Herstellen der Red-Black-Tree Eigenschaften.

Fazit: Eine Priority Queue kann auch als Red-Black Tree implementiert werden.

- MINIMUM geht beim MinHeap schneller als beim Red-Black Tree, nämlich in O(1).
- EXTRACT-MIN: Das Entfernen des Minimums dauert in beiden Fällen bzgl. O-Notation gleich lang, nämlich O(log n). Schließlich muss auch bei einem MinHeap bei EXTRACT-MIN erst wieder HEAPIFY aufgerufen werden.
- INSERT: In beiden Fällen O (log n).

Die asymptotischen Laufzeiten sind ähnlich, jedoch ist ein MinHeap bzw. MaxHeap bgzl. der Konstanten deutlich effizienter.

Aufgabe 2: Boyer-Moore

a)

Index	. . .	65	66	67	68	. . .	82	. . .
Arrayinhalt	-1	10	8	4	6	-1	9	-1
Bedeutung		A	B	C	D	…	R	…

b) 13 Zeichenvergleiche. Die folgende Illustration bezieht sich auf den Code der Vorlesung: BoyerMooreSearch.java. Die verglichenen Zeichen sind jeweils farbig, der Mismatch jeweils rot.

```
i = 0 → 3 Vergleiche (Fall 2a)
Index:   0 1 2 3 4 5 6 7 8 9 …
Text:    G C A A T G C C T A T G G G C T A T G T G
Muster:  T A T G T G

Dann i = 2 → 1 Vergleich (Fall 1)
Text:    G C A A T G C C T A T G G G C T A T G T G
Muster:      T A T G T G

Dann i = 8 → 2 Vergleiche (Fall 2b)
Text:    G C A A T G C C T A T G G G C T A T G T G
Muster:                  T A T G T G

Dann i = 9 → 1 Vergleich (Fall 1)
Text:    G C A A T G C C T A T G G G C T A T G T G
Muster:                    T A T G T G

Dann i = 15 → 6 Vergleiche (Match)
Text:    G C A A T G C C T A T G G G C T A T G T G
Muster:                                T A T G T G
```

c) Im folgenden Fall wird immer das ganze Muster mit dem Text verglichen. Beim Finden des Mismatches kann dennoch nur um 1 Stelle nach rechts geschoben werden.

```
Text:    A A A A A A A A A A A A A … A
Muster:  B A A A A
```

Milderung versprechen weitere Heuristiken, die aber nicht Bestandteil dieser Vorlesung sind:
http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/bmen.htm

Aufgabe 3: Rabin-Karp

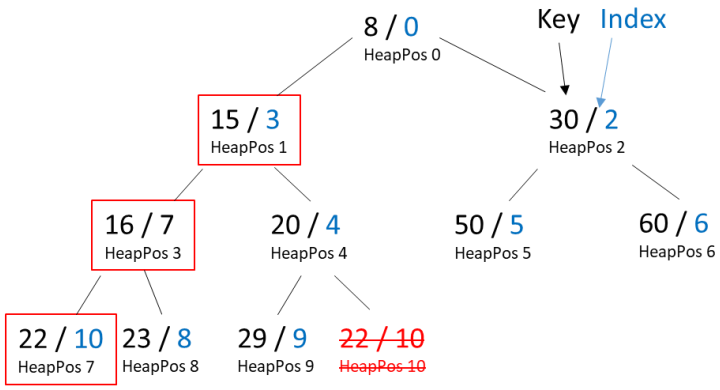
a) Ein Monte-Carlo-Algorithmus liefert nur mit hoher Wahrscheinlichkeit ein korrektes Ergebnis, ein Las-Vegas Algorithmus dagegen immer das richtige Ergebnis. Beide sind Varianten von randomisierten Algorithmen, die mit zufälligen Zwischenergebnissen (Zufallsgenerator) arbeiten. Im konkreten Fall (Rabin-Karp) wird sehr oft der Modulwert zufällig / durch einen Zufallsgenerator bestimmt.

b) Der Hashwert des Musters ist 26 mod 11 = 4.
Beim Durchgehen des Textes (jeweils 2er benachbarter Zeichen) stellt man fest, dass die benachbarten Ziffern 1 5, 5 9, 9 2 und 26 ebenfalls den gleichen Hashwert (bzgl. mod 11) ergeben.
Nur der letzte Vergleich mit 2 6 liefert einen Match. Also gibt es 3 falsche Treffer.

c) Man müsste für alle Muster den Hashwert vorberechnen. Bei jedem Weiterrücken im Text müsste man den aktuellen Text-Hashwert mit jedem Muster-Hashwert vergleichen.
Hinweis (kein Stoff): Damit der letzte Vergleich gegen mehrere Strings nicht zu teuer wird, benötigt man sogenannte Tries: https://en.wikipedia.org/wiki/Trie.

Aufgabe 4: Implementierung einer indizierten Priority Queue

Beim Löschen der 9 (Index 1) wird zunächst die Zahl ganz rechts im Heaparray 22 (Index 10) an dessen Stelle gesetzt. Anschließend rekursives `minHeapify`. Die Datenstrukturen sehen dann wie folgt aus:



pq	0	3	2	7	4	5	6	10	8	9	

heapPos	0	-1	2	1	4	5	6	3	8	9	7
keys	8	null	30	15	20	50	60	16	23	29	22
Index	0	1	2	3	4	5	6	7	8	9	10

Die Implementierung der beiden Funktionen gestaltet sich wie folgt:

```
// delete element with index i
public void delete(int i) {
    if (!contains(i)) throw new NoSuchElementException("index is not in the
priority queue");
    int pos = heapPos[i];
    exchange(pos, --n);
    swim(pos);
    minHeapify(pos);
    keys[i] = null;
    heapPos[i] = -1;
}

// change value of element with index i
public void changeKey(int i, Key key) {
    if (!contains(i)) throw new NoSuchElementException("index is not in the
priority queue");
    keys[i] = key;
    swim(heapPos[i]);
    minHeapify(heapPos[i]);
}
```