5. Priority Queues

5.1. Motivation

Priority Queues – oder Prioritätswarteschlangen – erlauben Elemente beliebig einzufügen und effizient das jeweils kleinste bzw. grösste Element zu lesen und zu entfernen. Solche Datenstrukturen werden zum Beispiel im Kern von Computer Betriebssystemen oder für Algorithmen auf Graphen benötigt. Eine verbreitete Implementierung von Priority Queues sind Array-basierte Heaps. Als spezielle Anwendung lässt sich damit ein Sortieralgorithmus mit interessantem Worst Case-Verhalten konstruieren.

5.2. Lernziele

- Sie können erklären was eine Priority Queue leistet und eine geeignete Schnittstelle formulieren.
- Sie können erklären, wie Heaps funktionieren und Heap-Operationen auf graphischen Darstellungen ausführen.
- Sie können einen Heap auf der Basis von einem Array in Java implementieren.
- Sie können *HeapSort* in Java implementieren und die asymptotischen Komplexitätsklassen des Laufzeitverhaltens angeben.

5.3. Priority Queues

Queues werden häufig benötigt, um Elemente „zwischenzulagern“, bis sie weiterverarbeitet werden können. Computer-Betriebssysteme verwalten in solchen Queues z.B. verschiedene Prozesse, die reihum ausgeführt werden, oder Events aus dem Benutzerinterface, um sie nach und nach zu verarbeiten.

Im Grunde funktionieren diese Queues wie die Warteschlagen an der Kinokasse. Neue Elemente werden hinten angefügt. Herausgenommen wird jeweils das vorderste Element.

In anderen Anwendungsfällen soll die Reihenfolge, in der die Elemente aus der Queue genommen werden, nicht einfach davon abhängen, wann sie eingefügt wurden, sondern von einem Schlüssel der Elemente. Ein solcher Schlüssel, für den eine Ordnungsrelation definiert sein muss, wird dann auch *Priorität* genannt.

Im Betriebssystem gibt es Prozesse, die mit Vorrang gegenüber anderen zum Zug kommen sollen, z.B. weil sie besonders zeitkritische Aufgaben erledigen. In diesen Fällen braucht man *Priority Queues*.

Wir unterscheiden zwei Typen von Priority Queues, je nach dem ob der kleinste Schlüssel die höchste Priorität hat oder umgekehrt. Die Standardoperationen auf einer Priority Queue sind dann die Folgenden:

Minimum Priority Queue

min()	gibt das kleinste Element zurück ohne die Priority Queue zu verändern
deleteMin()	entfernt das kleinste Element und gibt es zurück
add(elem)	fügt ein Element elem hinzu
size()	gibt die Anzahl Elemente in der Priority Queue

Maximum Priority Queue

max()	gibt das grösste Element zurück ohne die Priority Queue zu verändern
deleteMax()	entfernt das grösste Element und gibt es zurück

und natürlich auch hier add(elem) und size().

Schnittstellenbeispiel

Für einen abstrakten Datentyp *Minimum Priority Queues* käme folgende einfache Schnittstelle in Frage:

```
public interface MinPriorityQueue<K extends Comparable<? super K>> {
    void add(K element);
    K min();
    K removeMin();
    int size();
}
```

Passende Aufgaben: *Arbeitsblatt Aufgaben 1 und 2*

5.4. Implementierungsmöglichkeiten von Priority Queues mit bekannten Datenstrukturen

Priority Queues lassen sich mit verschiedenen mittlerweile bekannten Datenstrukturen implementieren.

Aufgabe: Füllen Sie die folgende Tabelle mit dem dem jeweiligen asymptotischen Laufzeitaufwand (in O-Notation) im Worst Case aus:

	Array		Linked List		Binärbaum	
Operation	**sortiert**	**unsortiert**	**sortiert**	**unsortiert**	**allgemein**	**AVL**
add(element)						
min()						
removeMin()						

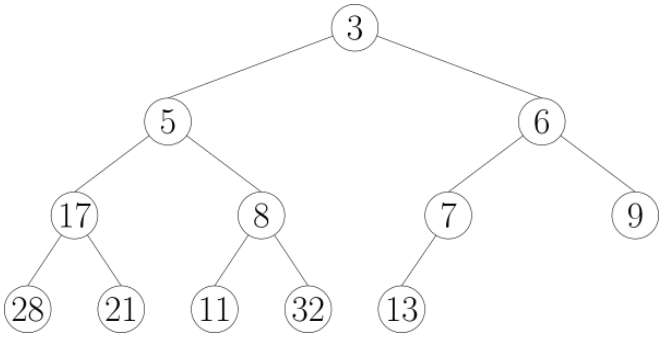
Tip: Bei den sequenziellen Datenstrukturen *Array* und *Liste* hat man eigentlich nur die Wahl, entweder die Elemente beim Einfügen mit entsprechendem Aufwand vollständig zu ordnen, um dann schnell das Minimum suchen und entfernen zu können, oder die Elemente ohne zu ordnen schnell einzufügen um dann mit entsprechendem Aufwand das Minimum zu bestimmen.

5.5. Datenstruktur Heap

Ein Heap sind eine Datenstruktur, die speziell für Priority Queues geeignet ist.

Konzeptuell sind Heaps binäre Bäume mit zwei weiteren invarianten Eigenschaften. Eine davon bezieht sich auf die *Struktur* des Baums, die zweite ist eine *Ordnungseigenschaft* zwischen den Elementen.

Nebenstehende Abbildung zeigt einen typischen *Min-Heap*.[1]



Struktur-Eigenschaft:
Vollständiger Binärbaum, mit Ausnahme der untersten Stufe; dort ist er von links nach rechts aufgefüllt.

Ordnungs-Eigenschaft (Min-Heap):
Der Schlüssel jedes Knotens ist kleiner gleich der Schlüssel seiner beider Kinder (falls vorhanden).

Ordnungs-Eigenschaft (Max-Heap):
Der Schlüssel jedes Knotens ist grösser gleich der Schlüssel seiner beider Kinder (falls vorhanden).

Passende Aufgaben: *Arbeitsblatt Aufgaben 3 und 4.*

5.6. Operationen auf (Min-)Heaps

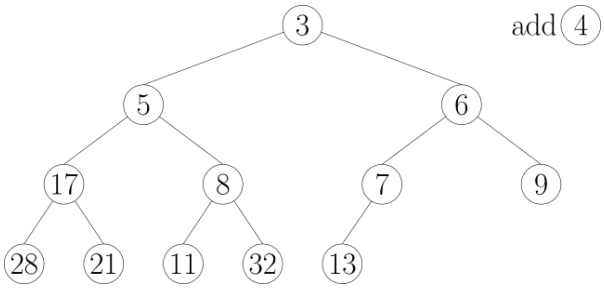
min():

Die Ordnungseigenschaft des Min-Heaps impliziert, dass das kleinste Element direkt an der Wurzel liegt.

Der asymptotische Aufwand hierfür ist: O(1) – Auf die Wurzel des Heaps kann direkt zugegriffen werden.

[1] Max-Heaps funktionieren genauso, nur sind die Elemente umgekehrt geordnet.

add(element):

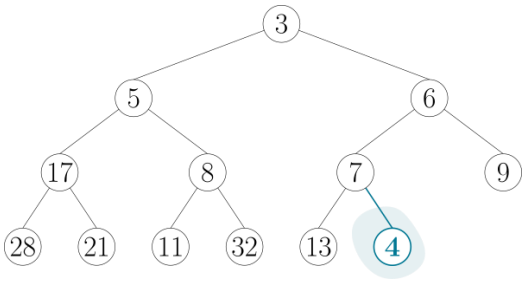


Ein neues Element muss so hinzugefügt werden, dass anschliessend sowohl die Struktur- als auch die Ordnungs-Eigenschaft des Heaps wieder gilt.

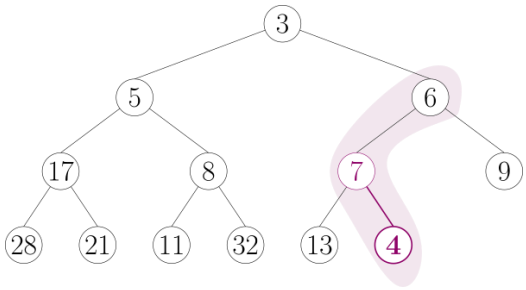
1. ***Struktur*-Eigenschaft:** Das neue Element wird an der ersten freien Stelle auf der untersten Stufe eingefügt, bzw. ganz links auf einer neuen Stufe, wenn der Baum vollständig ist.
2. ***Ordnungs*-Eigenschaft**: Solange das neu eingefügte Element „kleiner“ ist, als sein jeweiliger „Vater-Knoten“, wird es mit diesem vertauscht. So steigt das neue Element soweit im Heap nach oben, bis die Ordnungs-Eigenschaft gilt.

Der asymptotische Aufwand hierfür ist: O(log n) – Das Einfügen an die erste freie Stelle erfolgt in O(1) Zeit, wenn man sich diese merkt, sonst maximal O(log n), was der Höhe des Baumes entspricht. Das Aufsteigen erfolgt auf dem Pfad vom Blatt zur Wurzel, es gibt also maximal O(log n) Vertauschungen, wobei eine Vertauschung von zwei Elementen in O(1) erfolgt.

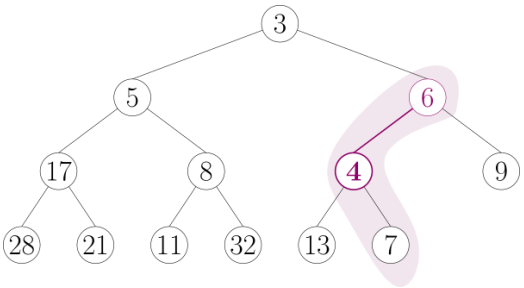
Beispiel:



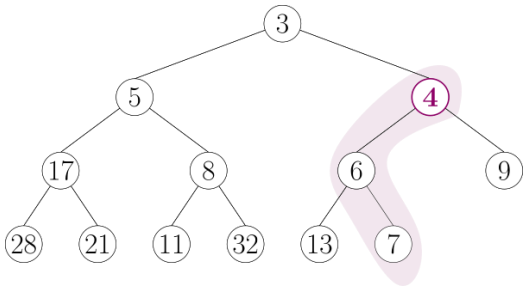
Schritt 1:
Die 4 wird an die erste leere Stelle eingefügt. Damit gilt die Struktureigenschaft bereits wieder.



Schritt 2a:
Die 4 ist kleiner als ihr Vater, d.h. die Ordnungseigenschaft ist verletzt. Die beiden werden darum vertauscht.



Schritt 2b:
Nach dem Tausch ist die 4 noch immer kleiner als der neue Vater. Darum wird wieder getauscht.



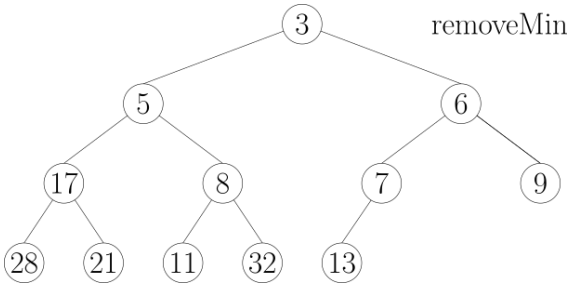
Schritt 2c:
Nun gilt die Ordnungseigenschaft wieder.

Beachte: Die Struktur ändert sich nach dem Einfügen (Schritt 1) nicht mehr, nur die Ordnung wird noch verändert.

removeMin():

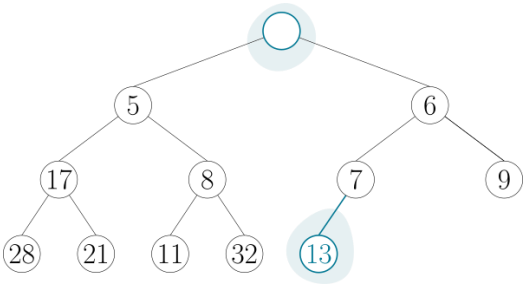
Das Minimum befindet sich in der Wurzel.

1. ***Struktur*-Eigenschaft:** Das Wurzelelement wird durch das letzte Element aus der untersten Stufe ersetzt.
2. ***Ordnungs*-Eigenschaft:** Das Wurzelelement so lange „hinuntersickern" lassen – d.h. es mit dem „kleineren" seiner Nachfolger vertauschen – bis es keinen „kleineren" Nachfolger mehr hat.

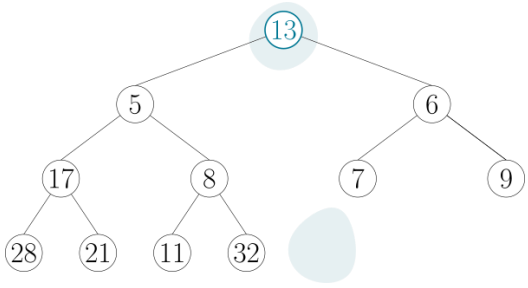


Der asymptotische Aufwand hierfür ist: O(log n) – Das ersetzen geht in O(1) Aufwand. Das Hinuntersickern erfolgt entlang eines Pfads von der Wurzel in ein Blatt. Es gibt also maximal O(log n) Vertauschungen, wobei eine Vertauschung in konstanter Zeit erfolgt.

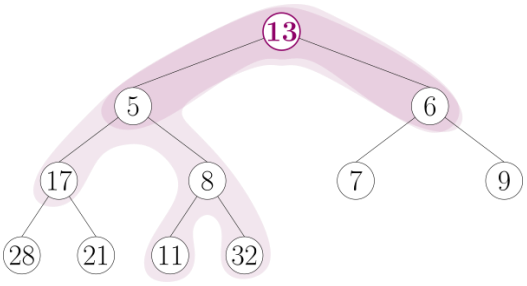
Beispiel:



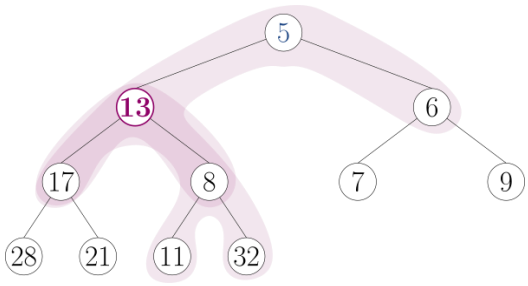
Schritt 1a:
Das Element in der Wurzel wird gelöscht ersetzt durch das letzte Element im Heap.



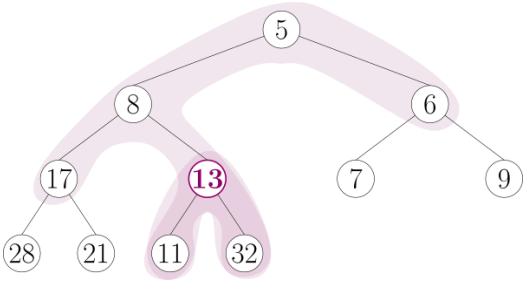
Schritt 1b:
Die Grösse des Heaps wird dadurch um eins kleiner. Die Struktureigenschaft gilt wieder.



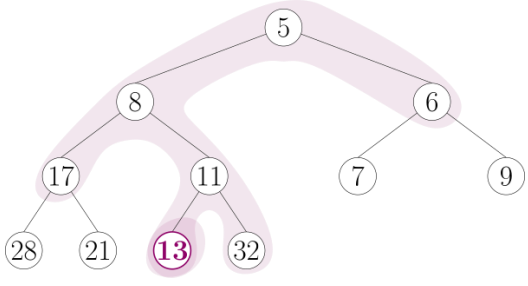
Schritt 2a:
Die Ordnungseigenschaft ist verletzt, weil die 13 grösser ist als ihre Kinder. Die 13 wird darum mit dem kleineren Kind vertauscht.



Schritt 2b:
Die 13 ist noch immer grösser als eines ihrer Kinder. Darum wird wieder mit dem kleineren Kind die Position getauscht.



Schritt 2c:
Gleiches Spiel: Die 13 und 11 verletzen die Ordnungseigenschaft und werden vertauscht.

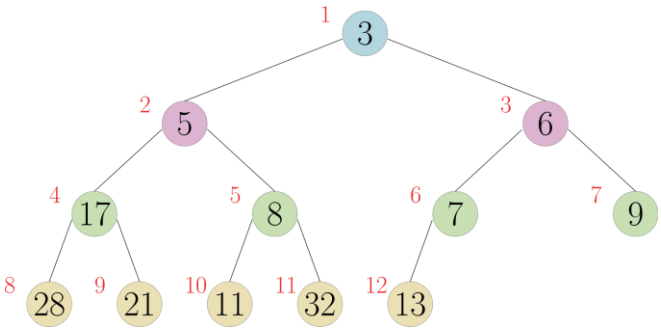


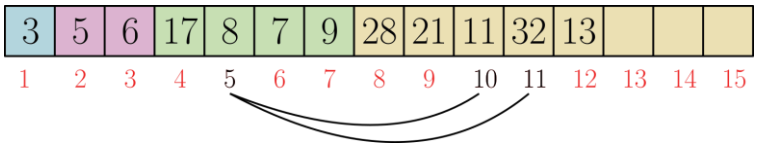
Schritt 2d:
Nun ist die Ordnungseigenschaft wieder hergestellt.

***Passende Aufgabe*:** *Arbeitsblatt Aufgabe 5*

5.7. Implementierung im Array

Eine interessante Eigenschaft von Heaps ist, dass man sie einfach und elegant in Arrays implementieren kann. Dazu legt man den Baum Stufe für Stufe hintereinander im Array ab:





Aufgabe: Die jeweiligen „Vater"- bzw. „Nachfolgerknoten" eines Knotens mit Index *i* findet man mittels Indexberechnungen. Füllen Sie die folgende Tabelle aus für 1-basierte und 0-basierte Indizes.

	Wurzel bei Index 1	Wurzel bei Index 0
Vater-Knoten von *i*		
linker Nachfolger von *i*		
rechter Nachfolger von *i*		
Indizes aller Blätter	bis	bis

Da sich bei den Programmiersprachen die Array-Indizierung ab 0 durchgesetzt hat, lohnt es sich zu überlegen, wie man diese Berechnungen für solche Arrays anpassen muss. Eine einfache Regel dafür ist:

Sei $f_1(x)$ die Berechnungsvorschrift für den Index eines Vorgänger-/Nachfolgers des Knotens mit Index x für ab 1 indizierte Arrays. Dann ist die entsprechende Berechnungsvorschrift $f_0(x)$ für ab 0 indizierte Arrays: $f_0(x) = f_1(x+1)-1$.

Operationen

min(): Gibt das erste Element des Arrays zurück

Zur Implementierung von *add* und *removeMin* werden die zwei Hilfsmethoden *siftUp (Hinaufsteigen)* und *siftDown* (Hinuntersickern) verwendet.

add(element):

1. Das neue Element *element* wird im ersten freien Platz im Array eingefügt.
2. Das neue Element wandert hinauf.

Beispiel in Java (0-basierte Indizes):

```
public void add(K element ) {
    heap[size] = element;
    heap.siftUp(size);
                size++;
}
```

removeMin():

1. Das erste Element im Array wird in lokaler Variable für die Rückgabe gespeichert.
2. Das letzte Element im Array wird an die erste Stelle geschoben.
3. Das neue erste Element sickert hinunter.

Beispiel in Java :

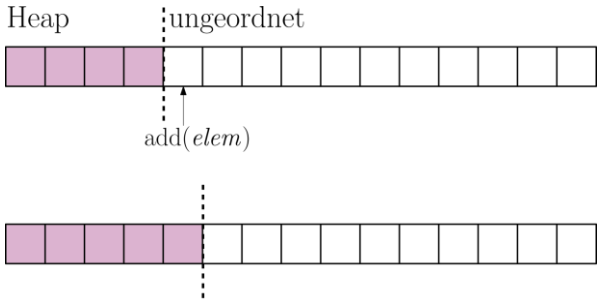
```
public K removeMin() {
    K res = heap.min();
    heap[0] = heap[size-1];
    heap[size-1] = null;
    size--;
    heap.siftDown(0);
    return res;
}
```

Passende Aufgaben: *Programmieraufgabe 1, Arbeitsblatt Aufgabe 6*

5.8. Aufbau eines Heaps in einem gefüllten Array

Wenn die Elemente bereits in einem Array enthalten sind, kann man den Heap an Ort und Stelle aufbauen. Da der Array bereits die Struktureigenschaft erfüllt, muss im Prinzip nur die Ordnungseigenschaft hergestellt werden.

Erste Idee: Heap *von vorne nach hinten* bauen (im Baum Top-Down). Wir stellen uns vor, dass wir die Elemente nacheinander in den Heap einfügen.

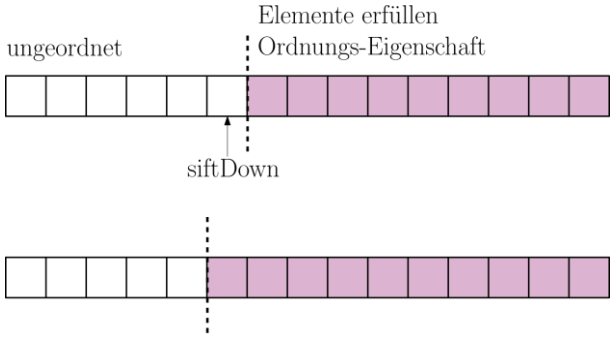


Aufwand: O(n log n) – wir rufen n Mal siftUp auf, welches jeweils bis zu log n Schritte benötigt.

Etwas genauer: Das siftUp benötigt log i Schritte, wenn i die Anzahl Elemente sind, die bereits zum Heap hinzugefügt wurden. Für die zweite Hälfte der Elemente ist das also mindestens log n/2. Zusammengezählt erhalten wir eine Laufzeit von mehr als n/2 * O(log n/2), was in O(log n) liegt.

Mit diesem Ansatz konnten wir also nicht ausnutzen, dass die Struktureigenschaft bereits hält.

Zweite Idee[2]: Heap *von hinten nach vorne* bauen (im Baum Bottom-Up). Wir nutzen aus, dass alle Blattknoten bereits die Ordnungs-Eigenschaft erfüllen, da sie haben keinen Nachfolger haben. Die inneren Knoten durchlaufen wir rückwärts (Bottom-Up) und lassen sie versickern.



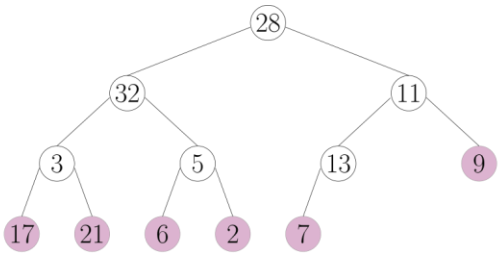
Wenn wir uns den Heap als Baum vorstellen, fällt es einfacher zu verstehen warum dies funktioniert:
Zu Beginn ist jedes Blatt ein (Teil-)Heap. Das Versickern eines inneren Knoten vereint die zwei Teilheaps seiner Kinder zu einem grösseren Teilheap. (Veranschaulichung auf der nächsten Seite.)

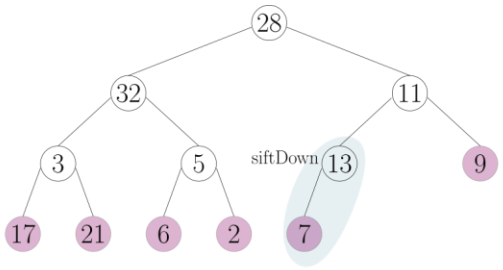
Aufwand: O(n)

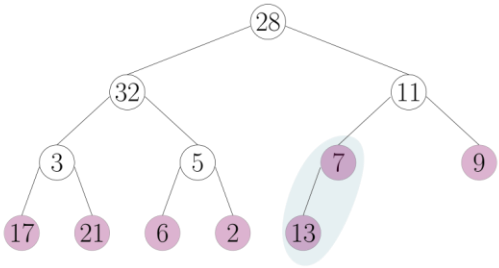
- Ein siftDown in einem Teilbaum mit Höhe i dauert O(i).
- Die Anzahl innerer Knoten auf Höhe i mit $2 \leq i \leq log n$ ist $n/2^i$.
- Zusammengerechnet erhalten wir $\sum_{i=2}^{log(n)} \frac{n}{2^i} \cdot O(i) = n \cdot O\left(\sum_{i=2}^{log(n)} \frac{i}{2^i}\right) \leqslant n \cdot O(2) = O(n)$.

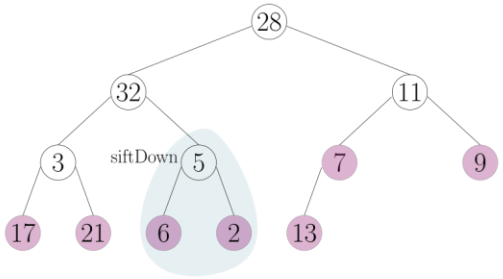
  Mit dem zweiten Ansatz konnten wir also ausnutzen, dass die Elemente bereits alle vorhanden sind. Schneller lässt sich ein Heap aus einer ungeordneten Menge an Elementen nicht bauen, weil jeder innere Knoten mindestens einmal auf die Ordnungseigenschaft überprüft werden muss / oder anders gesagt: weil jedes Element mindestens einmal betrachtet werden muss.

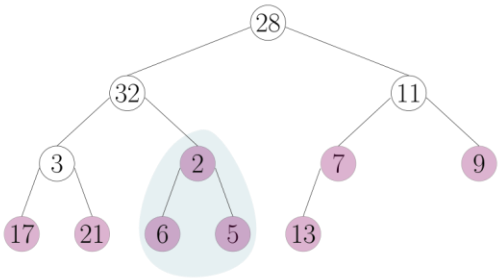
[2] Das hier beschriebene Verfahren ist auch als *Algorithmus von Floyd* bekannt. Robert W. Floyd und J. W. J. Williams gelten als die *Väter* von *HeapSort*.

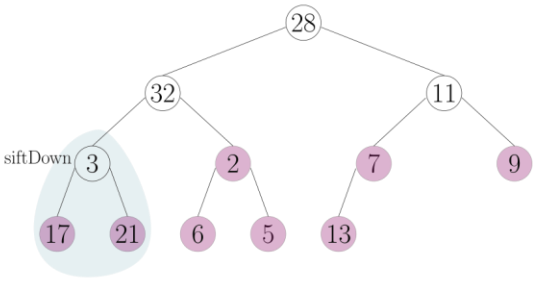


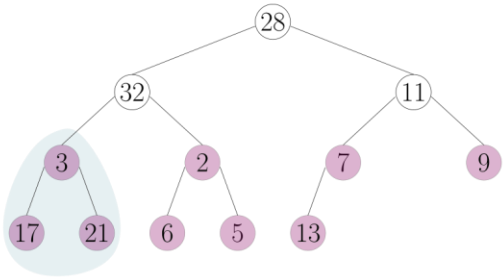


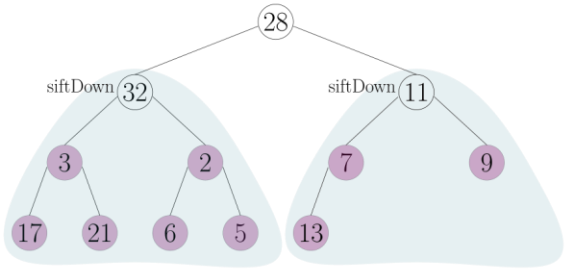


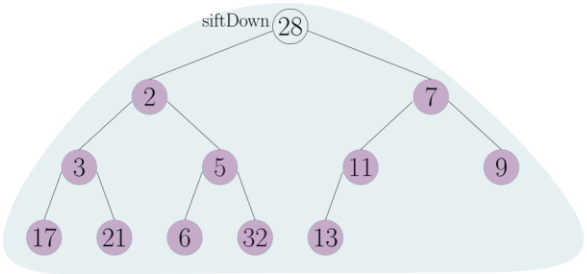


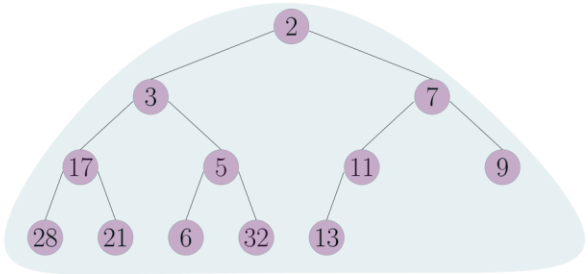












Passende Aufgaben: *Arbeitsblatt Aufgabe 7*

5.9. HeapSort

Mit Priority Queues lässt sich ein effizientes Sortierverfahren implementieren, das in zwei Phasen abläuft:

Phase 1: Alle Elemente werden in eine Priority Queue eingefügt

Phase 2: Die Elemente werden in geordneter Reihenfolge aus der Priority Queue herausgelesen

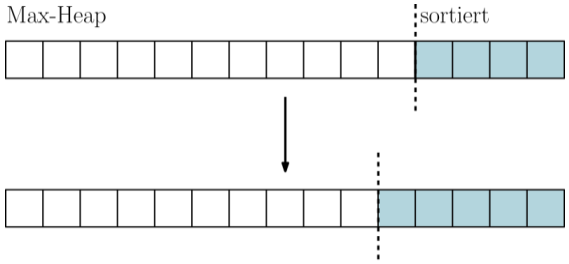
Der asymptotische Aufwand hierfür ist: O(n log n) – Jedes Einfügen dauert im Idealfall O(log n) und jedes herauslesen noch einmal O(log n). Ein Heap erreicht genau diese Laufzeiten. Dies ist auch sogleich der Best Case, weil sich n Elemente in beliebiger Reihenfolge nicht schneller sortieren lassen.

(Suboptimales) Beispiel in Java

```
public int[] PQsort (int[] values ) {
    int[] sorted = new int[values.length] ;
    MinPQ q = new MinPQ(values);
    for (int i = 0; i < values.length; i++) {
           sorted[i] = q.deleteMin();
    }
    return sorted;
}
```

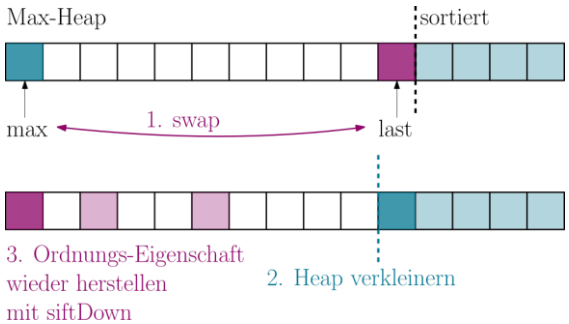
Unschön ist das obige Beispiel, weil die *Priority Queue* separaten Speicher in der Grössenordnung *O*(n) beansprucht. Da *Heaps* in einem Array abgelegt werden können, kann man damit ein Verfahren bauen, das ohne zusätzlichen Speicher auskommt. Dieses Verfahren nennt sicht *HeapSort*.

In Phase 1 steht das ganze Array für die Priority Queue zur Verfügung. In Phase 2 wird diese sukzessive kleiner und damit wird Platz frei für die fertig sortierte Folge von Elementen. Das Array wird also in zwei Bereiche geteilt. Einen davon belegt der Heap, den anderen die fertig sortierten Daten:



Die Wurzel des Heap liegt idealerweise bei Index 0, d.h., der Heap belegt den Bereich 0 bis n-k, wenn bereits k Elemente sortiert sind. An der Wurzel des Heaps sollte also jeweils das grösste Element zu finden sein, d.h., es wird ein Max-Heap benötigt.

Die in der Phase 2 in einer Schleife ablaufenden Schritte lassen sich so darstellen:



Da die Elemente vor dem Sortieren bereits alle im Array enthalten sind, kann man den Heap mit dem Algorithmus von Floyd in linearer Zeit bauen. Anschliessend wird n-1 Mal das grösste Element aus dem Heap extrahiert und im sortierten Teil eingefügt. Die Schritte benötigen jeweils O(1) für Swap und Heap verkleinern und O(log n) fürs Hinuntersickern der neuen Wurzel. Die Zeitkomplexität von HeapSort ist also O(n log n).

Weil HeapSort keinen zusätzlichen Speicher für die Elemente benötigt, sagt man das Verfahren ist *in place* oder *in situ.*

Passende Aufgaben: *Arbeitsblatt Aufgabe 7, Programmieraufgabe 2*