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 beispielsweise 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*.

Schnittstellenbeispiel

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

```
public interface MinPriorityQueue<K extends Comparable<? super K>> {
  void add(K element);   // fügt Element hinzu
  K min();               // Wert des minimalen Elements (Queue wird nicht verändert)
  K removeMin();         // entfernt das derzeit minimale Element und gibt es zurück
  int size();            // Anzahl aktuell in der Queue gespeicherter Elemente
}
```

Anwendungsbeispiel

Bei der Analyse grosser Datenmengen muss man bisweilen aus sehr vielen Elementen (z.B. mehrere Millionen) wenige (z.B. 100) bedeutsame heraussuchen. Nehmen wir an, ein *Iterator* liefert nacheinander *M* Elemente, von denen die *N* mit den grössten Schlüsseln gesucht sind.

Ansatz 1: Die *M* Elemente werden in einer Liste gespeichert, sortiert, und dann werden die *N* grössten aus dieser Folge gelesen.

Der asymptotische Aufwand hierfür ist:

Ansatz 2: Während Element für Element aus dem *Iterator* gelesen wird, werden in einer Priority Queue die *N* grössten jeweils bereits gelesenen Elemente gesammelt. Für jedes neu gelesene Element wird geprüft, ob es grösser ist, als das kleinste in der Priority Queue. Nur dann muss es berücksichtigt werden.

Der asymptotische Aufwand hierfür ist abhängig vom Aufwand für die Operationen der Priority Queue und von der Verteilung der Daten im Datenstrom. Idealerweise liefert die Priority Queue jeweils das minimale Element mit Aufwand $O(1)$. Bei gleichverteilten Eingaben-Daten wird damit der Gesamtaufwand zu $O(M)$.

Ansatz 2 kann so in Java programmiert werden:

```
<K extends Comparable<? super K>> K[] maxN(Iterator<K> it, int N) {
 MinPriorityQueue<K> q = new ...;
 while (q.size() < N && it.hasNext()) q.add(it.next()); // Priority Queue füllen
 while (it.hasNext()) {









 }
 K[] arr = (K[])new Comparable[N];
 for (int i = 0; i < N; i++) arr[i] = q.removeMin();
 return arr;
}
```

5.4. Implementierungsmöglichkeiten von Priority Queues mit bekannten Datenstrukturen

Priority Queues lassen sich mit verschiedenen mittlerweile bekannten Datenstrukturen implementieren. Der asymptotische Laufzeitaufwand im Worst Case dafür ist jeweils:

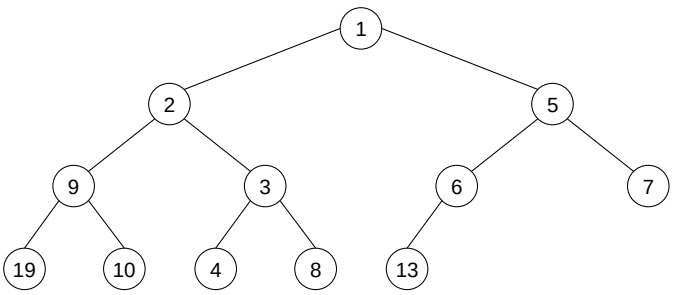
	Array		Linked List		Baum	
Operation	**sortiert**	**unsortiert**	**sortiert**	**unsortiert**	**allgemein**	**AVL**
add(element)						
min()						
removeMin()						

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



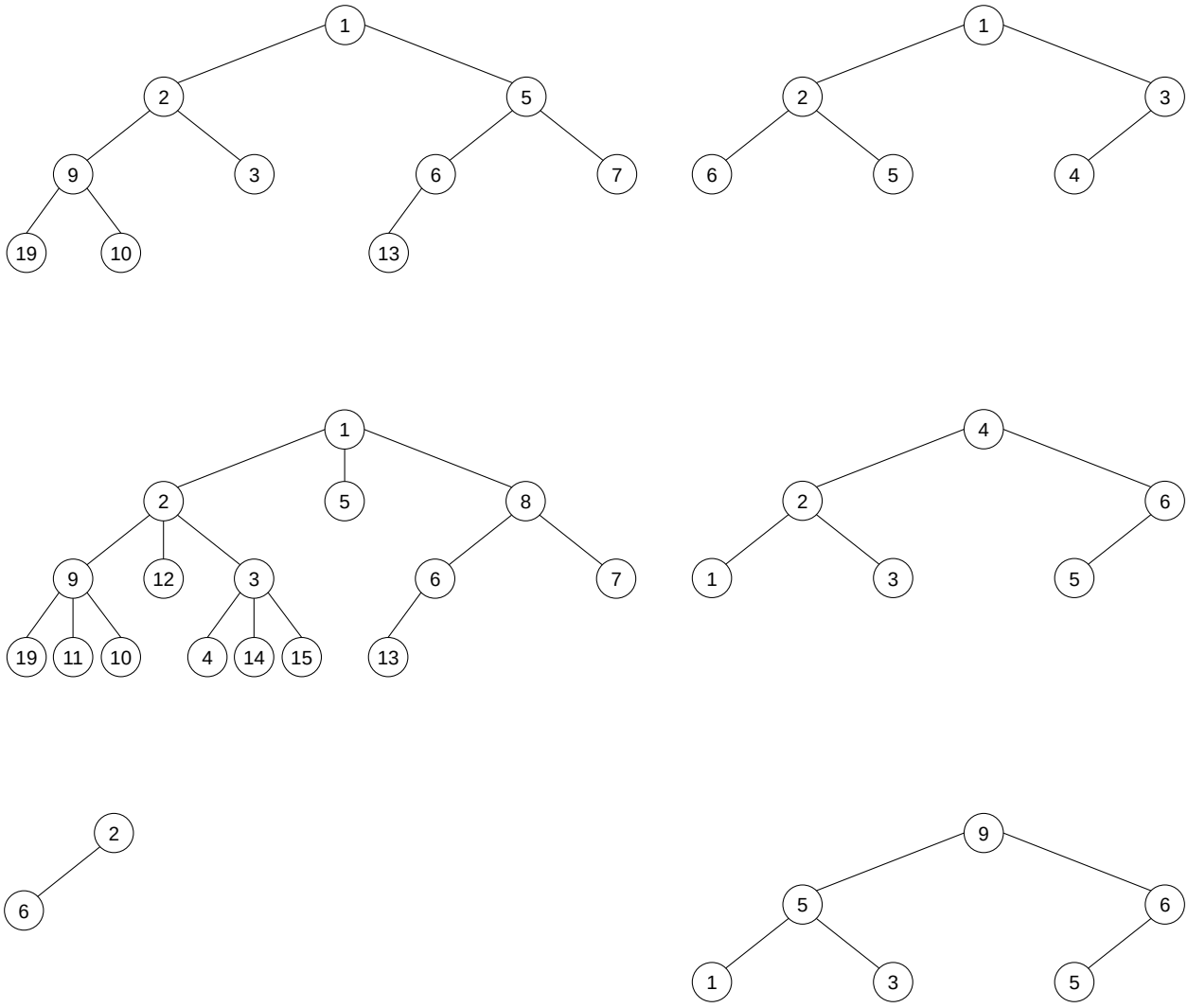
Nebenstehende Abbildung zeigt einen typischen *Min-Heap*.[1]

Struktur-Eigenschaft von Heaps:

Ordnungs-Eigenschaft von (Min-)Heaps:

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

Beispiele von Heaps und nicht-Heaps

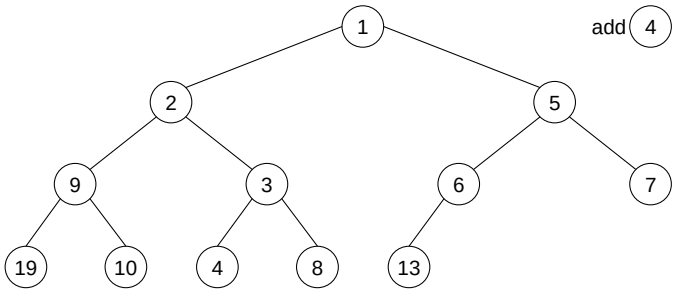


5.6. Operationen auf (Min-)Heaps

min(): Aus der Ordnungsinvariante des Min-Heap folgt, dass das minimale Element direkt an der Wurzel gefunden wird.

Der asymptotische Aufwand hierfür ist:

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

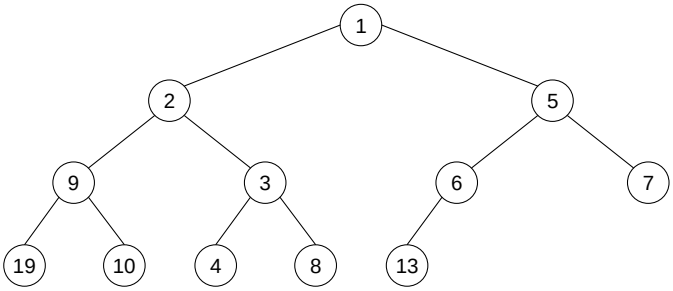


1. *Struktur*-Eigenschaft: Das neue Element wird an der ersten freien Stelle auf der untersten Stufe eingefügt, bzw. ganz links auf einer neuen Stufe, wenn der Baum vollständig ist.
2. *Ordnungs*-Eigenschaft: Solange das neu eingefügte Element „kleiner“ ist, als sein jeweiliger „Vater-Knoten“, wird es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

Der asymptotische Aufwand hierfür ist:

removeMin(): Das Minimum befindet sich in der Wurzel.

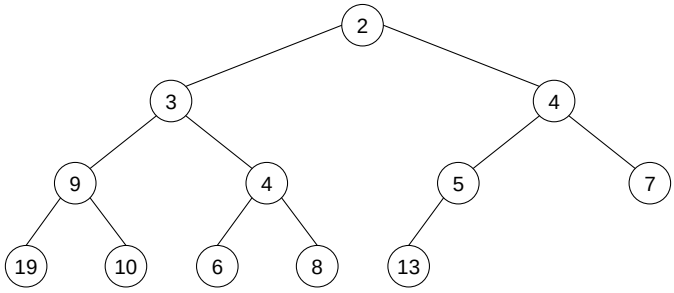
1. *Struktur*-Eigenschaft: Das Wurzelelement wird durch das letzte Element aus der untersten Stufe ersetzt.
2. *Ordnungs*-Eigenschaft: Das Wurzelelement so lange „hinuntersickern“ lassen – d.h. es mit dem „kleineren“ seiner Nachfolger vertauschen – bis es keinen „kleineren“ Nachfolger mehr hat.



Der asymptotische Aufwand hierfür ist:

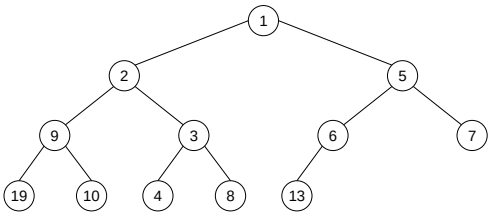
Beispiel

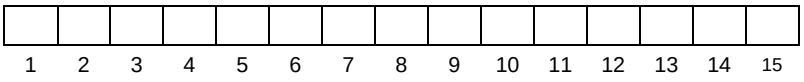
add(2);
removeMin();



5.7. Heap-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:





Die jeweiligen „Vater“- bzw. „Nachfolgerknoten“ eines Knotens mit Index *i* findet man mittels Indexberechnungen:

	Wurzel bei Index 1	Wurzel bei Index 0
Vater-Knoten von *i*		
linker Nachfolger von *i*		
rechter Nachfolger von *i*		

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 Hilfsmethoden *siftUp* und *siftDown* vorausgesetzt, die entsprechend das Hinaufsteigen oder Hinuntersickern eines Elements im Heap bewerkstelligen.

siftUp(index):

Solange *index* nicht die Wurzel bezeichnet und der Vater-Knoten des Knotens *index* „grösser ist“:

Vertausche den Knoten *index* mit seinem Vater-Knoten und setze *index* auf dessen Index

siftDown(index):

Wenn *index* zwei Nachfolger hat, wähle *j* als den Index des „kleineren“ der beiden, ansonsten wähle *j* als Index des linken Nachfolgers.

Solange an der Stelle *j* ein Element des Heaps ist und das Element *j* „kleiner ist“ als das Element *index*,

Vertausche den Knoten *j* mit seinem Vater-Knoten *index*,

setze *index* auf *j*

Wenn *index* zwei Nachfolger hat, wähle *j* als den Index des „kleineren“ der beiden, ansonsten wähle *j* als Index des linken Nachfolgers.

add(element):

1. Fügt *element* im ersten freien Platz im Array ein
2. siftUp(*Einfügeort*)

removeMin():

1. Speichert das erste Element im Array in lokaler Variable für Rückgabe
2. Verschiebt das letzte Element des Arrays an die erste Stelle
3. siftDown(Index der Wurzel)

5.8. HeapSort

Mit Priority Queues kann man ein effizientes Sortierverfahren implementieren. Dieses läuft in zwei Phasen ab:

Phase 1: Alle Elemente werden gemäss Ihres Schlüssels in eine Priority Queue eingefügt

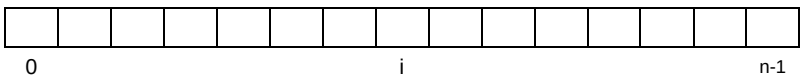
Phase 2: Die Elemente werden in geordneter Reihenfolge aus der Priority Queue herausgelesen

Der asymptotische Aufwand hierfür ist im Best Case:

Der asymptotische Aufwand hierfür ist im Worst Case:

Unschön ist es, wenn 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.

In Phase 1 steht das ganze Array für die Priority Queue zur Verfügung. In Phase 2 wird diese sukzessive kleiner und damit wird Platz 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:

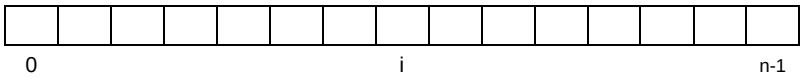


Die Wurzel des Heap liegt idealerweise bei Index

d.h., der Heap belegt den Bereich

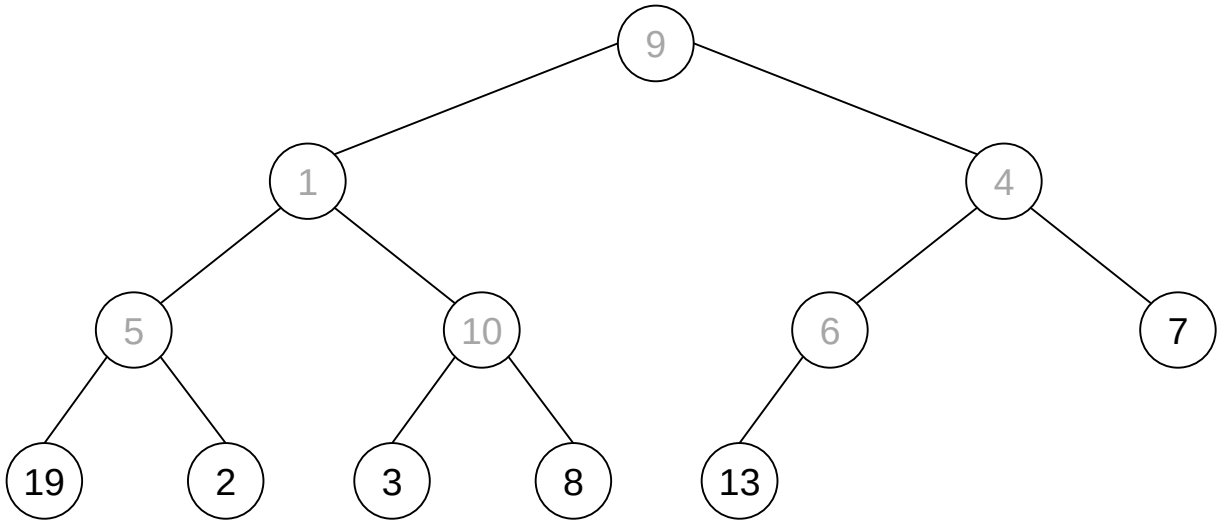
An der Wurzel des Heaps sollte also jeweils das Element zu finden sein, d.h., es wird ein -Heap benötigt.

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



Aufbau eines Heaps in einem gefüllten Array[2]

Da die Elemente vor dem Sortieren bereits alle im Array enthalten sind, kann man den Heap an Ort und Stelle „von unten nach oben“ aufbauen. Alle Blattknoten für sich betrachtet erfüllen bereits die Ordnungs-Eigenschaft des Heaps, denn sie haben keinen Nachfolger, der die Ordnung verletzen würde.



In einem Array mit *n* Elementen haben die Blätter die Indizes

von

bis

Nimmt man nun schrittweise rückwärts die restlichen Knoten dazu, muss die Ordnungs-Eigenschaft ggf. mittels eines *siftDown* des jeweils neu betrachteten Knoten etabliert werden.

[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*.