Theorie

1. Prioritäts-Warteschlangen

Die Wichtigkeit der beiden Datenstrukturen *Stack* (Stapel, LIFO-Queue) und *Queue* (Warteschlange, auch FIFO-Queue) dürfte hinreichend bekannt sein. Wir haben sie bisher auch als nützliche Werkzeuge für verschiedene algorithmische Aufgaben in der Informatik kennengelernt. Auch im täglichen Leben begegnen wir täglich Einrichtungen, die entweder als *Stack* (z.B. Tellerwärmer in einem Restaurant) oder als *Queue* organisiert sind (Warteschlange vor dem Geldautomaten).

Es gibt auch Situationen, in welchen die Elemente in einer Warteschlange *unterschiedliche Wichtigkeit* besitzen und deshalb nicht exakt in der Reihenfolge ihres Eintreffens, sondern bevorzugt behandelt werden sollen (Beispiel: Eine Schlange von Patienten steht vor der Notfallabteilung eines Spitals, und in neuer Patient kommt hinzu, welcher in akuter Lebensgefahr schwebt). In Betriebssystemen können Prozesse unterschiedliche Prioritäten besitzen; höher priorisierte Prozesse werden zeitlich bevorzugt behandelt. Um Organisationsformen dieser Art zu realisieren, werden häufig sogenannte *Prioritäts-Warteschlangen* eingesetzt, welche die eintreffenden Elemente automatisch nach Wichtigkeit und innerhalb gleicher Wichtigkeit chronologisch anordnen.

1.1 Operationen auf Prioritäts-Warteschlangen

Folgende Operationen sind für Prioritäts-Warteschlangen interessant:

- Zugriff und/oder Entfernen des Elements mit der höchsten Priorität.
- Einfügen eines neuen Elements mit gegebener Priorität. Es muss automatisch an der richtigen Stelle plaziert werden, d.h vor allen anderen Elementen mit niedrigerer Priorität, aber hinter allen Elementen mit gleicher oder höherer Priorität.
- Ändern der Priorität eines bereits existierenden Objekts (diese Operation wird zerlegt in zwei Operationen: 1) entfernen eines bestimmten Objekts x aus der Warteschlange. 2) Einfügen von x mit einer neuen Priorität).

Insgesamt müssen also 4 *Elementar-Operationen* implementiert werden, so dass sie möglichst effizient arbeiten:

a) Zugriff auf das Element mit höchster Priorität.
b) Entfernen des Elements mit höchster Priorität.
c) Einfügen eines Objekts mit gegebener Priorität.
d) Entfernen eines beliebigen Objekts mit bekannter Position.

1.2 Prioritäts-Warteschlangen mit sortierten Listen – eine Analyse

Bisher haben wir doppelt verkettete Listen benutzt, um Warteschlangen zu implementieren. Für eine Prioritäts-Warteschlange käme demnach vielleicht eine sortierte Liste in Frage. Wir wollen kurz untersuchen, wie effizient die geforderten Grundoperationen a) bis d) mit sortierten Listen realisierbar sind:

a) Element am Kopf der Liste abrufen. Sehr einfach. *Aufwand: O(1).*

b) Element vom Kopf der Liste entfernen. Sehr einfach. *Aufwand: O(1).*

c) Auch in einer *sortierten*, linearen Liste beträgt der Aufwand für die Suche der Einfügeposition O(N). Man gewinnt durch die Tatsache der Sortiertheit kaum einen Vorteil. Die Einfüge-Operation selbst ist dann wieder sehr einfach. Gesamter *Aufwand: O(N).*

d) Da die Position bekannt ist, erfordert das Löschen eines beliebigen Elements der doppelt verketteten Liste nur die Anpassung je einer Referenz im Vorgänger und im Nachfolger. *Aufwand: O(1).*

Da wir in Operation c) die Vorteile einer sortierten Struktur nicht ausnutzen können, bleibt der Aufwand unvertretbar hoch, während die anderen Operationen mit optimaler Geschwindigkeit ausgeführt werden. Wir müssen also eine andere Datenstruktur verwenden, welche eine geeigneteres Aufwandsprofil besitzt (ein Array kommt auch nicht in Frage, wie Sie bei einer Analyse der Operationen a) bis d) einfach selbst nachvollziehen können.

2. Heaps

Unter einem *Heap* (*dt: Halde*) versteht man einen balancierten binären Baum, für dessen Knoten die sogenannte *Heap-Bedingung* gilt. Beachten Sie folgende beide Punkte:

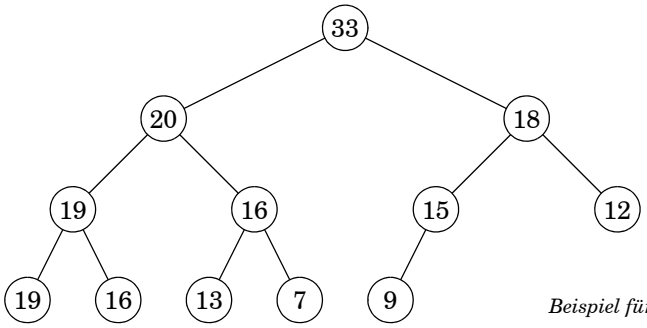
- Verwechseln Sie diese *Datenstruktur* Heap nicht mit dem Heap aus dem Gebiet der *Speicherverwaltung* (das ist der Ort, wo in Java Objekte und Arrays abgelegt werden). Diese beiden Begriffe haben nichts miteinander zu tun!
- Falls Sie aus der recht anspruchsvollen Übung mit balancierten AVL-Bäumen noch ungute Ressentiments mitbringen, können Sie beruhigt weiterlesen: Heaps haben nichts mit AVL-Bäumen zu tun und sind vergleichsweise einfach zu verwalten.

Ein binärer Baum heisst Heap, wenn folgende Bedingungen erfüllt sind:

a) Der Baum ist etagenweise vollständig aufgefüllt. Das heisst: Bei einem Baum der Höhe h sind die Etagen *1* bis *h–1* gänzlich gefüllt, während die Etage h unvollständig sein darf. Die Knoten der Etage h müssen aber zusammenhängend am linken Ende des Baumes angeordnet sein.

b) Für jeden Knoten k gilt: $k.key \geq k.L.key$ und $k.key \geq k.R.key$. Oder anders gesagt: Der Schlüssel von k ist mindestens so gross die der grösste Schlüssel seiner Söhne. Zwischen den Schlüsseln der Söhne ist allerdings, im Unterschied zum bekannten Suchbaum aus Übung 4, keine Bedingung definiert!

Die unter b) genannte Bedingung definiert einen sogenannten *Max-Heap*, bei dem das *grösste* Element garantiert in der Wurzel des Baumes zu finden ist. Sie kann auch abgeändert werden, indem alle «≥» durch «≤» ersetzt werden: Dabei entsteht ein *Min-Heap* mit dem kleinsten Element in der Wurzel.



Beispiel für einen Max-Heap

Im Beispiel oben können wir erkennen, dass in einem Heap keine absolute Sortier-Reihenfolge besteht. Insbesondere ist es nicht möglich, wie bei einem Suchbaum, einen beliebigen Knoten mit logarithmischer Suchzeit zu finden.

Wir wollen nun überprüfen, ob sich die Datenstruktur Heap für die Implementierung einer Prioritäts-Warteschlange eignet, indem wir untersuchen, *wie* und vor allem *wie schnell* die Operationen a) bis d) aus Abschnitt 1.1 auf einem Heap ausgeführt werden können.

3. Grundoperationen auf einem Heap

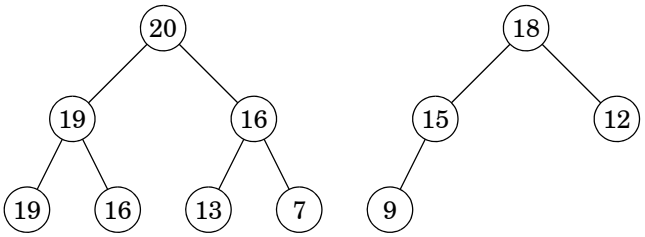
3.a Zugriff auf das Element höchster Priorität

Gemäss Heap-Bedingung kann kein Knoten grösser sein als die Wurzel. Demzufolge ist das grösste Element durch einen einfachen Zugriff auf die Wurzel erreichbar. *Aufwand: O(1)*

3.b Entfernen des Elements mit höchster Priorität

Entfernt man das Element mit der höchsten Priorität (die Wurzel), so zerfällt der Heap in die beiden Teilbäume der Wurzel, das heisst, in zwei separate Heaps. Sie müssen nun wieder zu einem einzigen Heap zusammenge-

fügt werden, damit die Datenstruktur wieder ordentlich benutzt werden kann. Das *Entfernen* des Objekts fällt mit *O(1)* sehr günstig aus, aber was kostet die Vereinigung der beiden Heaps zu einem einzigen?

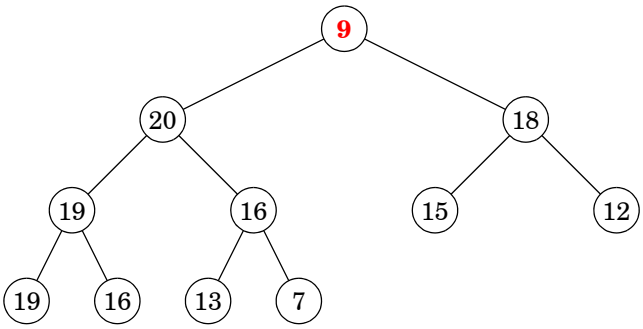


Vorgehensweise:

1. Anstelle der ehemaligen Wurzel wird der *letzte Knoten* des Baumes eingesetzt. Er liegt in der untersten Ebene ganz rechts.

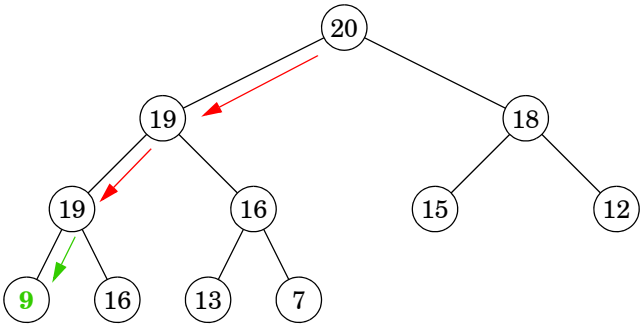
   Weil immer das letzte Element als Substitut gewählt wird, bleibt der Baum stets ausbalanciert.

   Allerdings ist jetzt die Heap-Bedingung nicht mehr erfüllt, denn der letzte Knoten im Baum ist normalerweise kleiner als die ehemalige Wurzel. Sie muss im nächsten Schritt wieder hergestellt werden.



2. Um die Heap-Bedingung wieder herzustellen, wird der neu plazierte Knoten k mit seinen Söhnen verglichen. Falls mindestens ein Sohn grösser ist, wird k mit dem grösseren der beiden Söhne vertauscht. Dieser Vorgang wird mit k so lange wiederholt, bis er an einen Ort gelangt, an welchem die Heap-Bedingung nicht mehr verletzt. Diese Verfahrensweise wird als ***Versickern nach unten*** (*sift down*) bezeichnet.

   Merke: Für einen Knoten *ohne* Söhne gilt die Heap-Bedingung *immer* als erfüllt.



Laufzeit dieser Operation:

Da mit jeder Iteration eine Stufe des Baumes überwunden wird und der Baum aufgrund seiner hervorragenden Balance höchstens $log_2(N)+1$ Stufen besitzt, sind maximal *O(log N)* Operationen erforderlich.

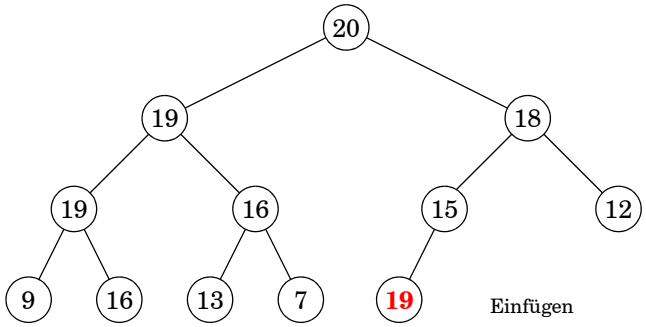
3.c Einfügen eines Objekts mit gegebener Priorität

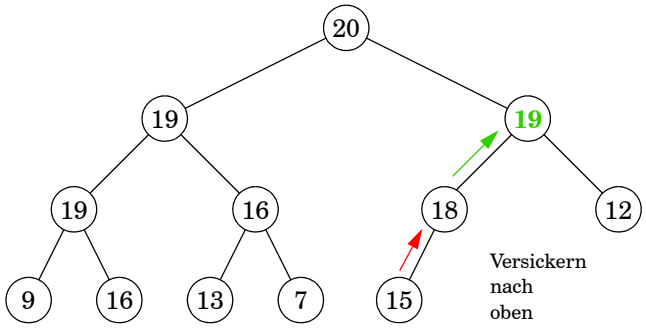
In einer linearen Liste müssten wir für die Einfüge-Operation zuerst die passende Einfügestelle suchen, was bekanntlich nur mit linearem Aufwand erfolgen kann.

Für das Einfügen eines Elements in den Heap wählen wir aber ein anderes Verfahren:

1. Zunächst wird der neue Knoten k als *letzter* Knoten in den Baum eingefügt.
2. Da über die Priorität von k keine Aussage getroffen werden kann, muss angenommen werden, dass der Vater von k in manchen Fällen die Heap-Bedingung nun nicht mehr erfüllt. k muss nun also mit seinem Vater verglichen werden: Falls der Schlüssel von k grösser ist als der des Vaters, werden die beiden Knoten vertauscht. Dieser Vorgang wird auf dem Pfad zur Wurzel so lange fortgesetzt, bis die Heap-Bedingung nicht mehr verletzt wird. Wir nennen diese Iteration ***Versickern nach oben*** (*sift up*).

Die beiden Schritte am Beispiel «*Knoten mit der Priorität 19 einfügen*» :





Laufzeit dieser Operation:

Zwei Operationen sind nötig, um das vollständige Einfügen zu vollziehen:

Suchen der Einfügestelle. Dies ist mit logarithmischem Aufwand möglich. Das Verfahren hierzu sowie den Beweis, dass der von mir behauptete Aufwand stimmt, müssen Sie in einer Übungsaufgabe aber selbst erbringen.

Plazieren des neuen Knotens an der richtigen Stelle: Die Versickerung nach oben verläuft nach den selben Prinzipien wie das Versickern nach unten. Mit jedem Schritt wird eine Ebene erklommen. Somit beträgt der Aufwand ebenfalls *O(log N)*.

Beide Teil-Operationen beanspruchen je einen Aufwand von *O(log N)*, somit gilt dies auch für die gesamte Operation!

3.d Entfernen eines beliebigen Objekts mit bekannter Position

Wir setzen voraus, dass wir den zu löschenden Knoten gefunden haben und eine Referenz auf ihn besitzen. Wir wissen, dass wir alle Knoten, welche keine Söhne besitzen, ohne weitere Massnahmen entfernen können, da dadurch die Heap-Bedingung nie verletzt werden kann. Aus diesen Tatsachen ergibt sich eine Strategie zum Löschen eines beliebigen Knotens:

1. Der letzte Knoten im Baum wird auf den zu löschenden Knoten kopiert, welcher damit überschrieben wird.
2. Der letzte Knoten wird dann an seiner ursprünglichen Position gelöscht.
3. Der ehemals letzte Knoten k befindet sich jetzt an einer neuen Position und verletzt hier möglicherweise die Heap-Bedingung. Falls k einen grösseren Schlüssel als sein neuer Vater besitzt, verschieben wir ihn mit einem *sift-up* so weit nach oben wie nötig. Existiert mindestens ein Sohn von k, dessen Schlüssel grösser ist als der des Vaters, lassen wir k durch ein *sift-down* so weit wie nötig nach unten versickern.

Beide Versickerungs-Operationen besitzen logarithmische Komplexität, deshalb kann die Operation 3.d immer mit maximal logarithmischer Laufzeit ausgeführt werden.

♦

Zusammenfassend stellen wir fest: Keine der vier vorgestellten Operationen übersteigt die Komplexität *O(log N)*. Die Datenstruktur eignet sich somit für die Implementierung eines Heaps.

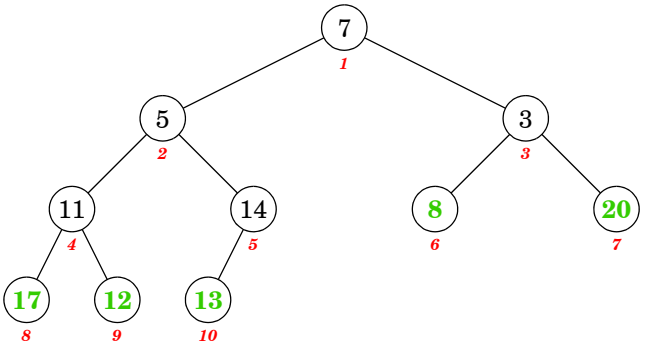
4. Einen Heap aufbauen

J. W. J. Williams, der Erfinder des *Heap-Sorts*, hat 1964 folgenden Algorithmus zum Ausbau eines Heaps vorgeschlagen: *«Beginne mit einem leeren Heap und füge jeden Knoten mit der Operation c) ein.»* Es ist klar, dass dieser Algorithmus funktioniert, denn auch ein leerer Heap ist ein gültiger Heap, und wir haben in 3.c gezeigt, dass durch die dort beschriebene Operation c) ein gültiger Heap in einen neuen, ebenfalls gültigen, Heap übergeführt wird.

Dieser Algorithmus eignet sich für Fälle, bei welchen die neuen Elemente tröpfchenweise eintreffen und deshalb *einzeln* eingefügt werden müssen (*sog. Online-Algorithmus*). Besteht hingegen bereits zu Beginn eine grössere Ansammlung von Elementen, kann aus ihnen auf wesentlich effizientere Weise ein Heap aufgebaut werden (*Algorithmus von R. W. Floyd, 1964*).

Algorithmus von Floyd

1. Baue aus den vorhandenen N Elementen einen nach der Heap-Regel a) einen balancierten binären Baum auf. Somit sind alle Ebenen ausser der untersten vollständig gefüllt, und die Elemente der untersten Ebene liegen kompakt am linken Ende.
2. Jetzt müssen die Knoten, welche die Heap-Bedingung verletzen, an die richtige Stelle gebracht werden. Knoten ohne Söhne verletzen die Heap-Bedingung nie, sie müssen deshalb nicht weiter beachtet werden (in der Graphik rechts grün markiert).



Die anderen Knoten (*innere Knoten*) müssen untersucht und nötigenfalls verschoben werden. Zur Verdeutlichung der weiteren Schritte führen wir eine fortlaufende Numerierung der Knoten ein, die bei Eins beginnt und etagenweise fortschreitet (rote Zahlen in der Graphik oben). Es ist einfach einzusehen, dass etwa die Hälfte aller Knoten die Heap-Bedingung nicht verletzen. Genauer: Nur die Knoten 1 bis N/2 (abgerundet, wie bei ganzzahliger Division) müssen überprüft werden. Sie werden alle nach unten versickert, wobei man beim Knoten N/2 beginnt und sich rückwärts bis zur Wurzel hocharbeitet.

Im Beispiel:

- Knoten 4 (*key=11*) wird versickert zur Pos. 8
- Knoten 3 (*key=3*) wird versickert zur Pos. 7
- Knoten 2 (*key=5*) wird versickert zur Pos. 9
- Knoten 1 (*key=7*) wird versickert zur Pos. 6