

HeapSort (J.W.J. Williams 1964)

Dieser Algorithmus zeigt, dass man durch eine kluge Strukturierung der Daten auf ganz neue Ideen gebracht werden kann. Wenn Du im Mathematikunterricht bereits von Graphen, Bäumen, Knoten und Kanten gehört hast, wird Dir vieles bekannt vorkommen.

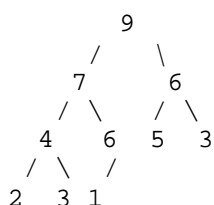
Die kluge Datenstruktur

ist ein 'binärer maximal-Heap'. (heap=Haufen).

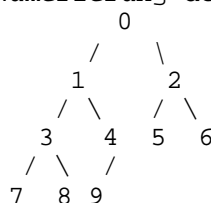
Trotz des komplizierten Namens ist er aber einfach nur ein Baum mit folgende Eigenschaften:

- ganz oben an der Wurzel des Heaps befindet sich ein einziger Knoten (Wert).
- von jedem Knoten gehen 0,1 oder 2 Verbindungen zu weiteren Knoten ('Kindern') aus.
- der Wert jedes Knotens ist größer als der Wert seiner Kinder.
- der Heap ist gleichmäßig gefüllt. Nur die letzte Zeile darf rechts teilweise leer bleiben.

Beispiel:



die Numerierung der Plätze:



die entsprechende Liste wäre also $L=[9,7,6,4,6,5,3,2,3,1]$

wie können wir hier navigieren: durch Veränderung der Platznummern

nach oben – den parent des Knotens Nummer n	<code>def parent(n): return (n-1)/2</code>
nach unten – das linke child	<code>def links(n): return 2*n+1</code>
nach unten – das rechte child	<code>def rechts(n): return 2*n+2</code>

ANGENOMMEN, eine zu sortierende Liste läge in dieser Form vor.

Eine Sortierung der Größe nach wäre nun nicht allzu kompliziert: das Element an der Wurzel ist sicher das größte von allen. Wir nehmen es heraus und stehen nun vor der Aufgaben, den Heap zu 'reparieren', um die gewünschten Eigenschaften zu erhalten. Wer muss jetzt ganz nach oben: sicher das größere der beiden Kindelemente. Schieben wir das hoch, so ist der Wurzelplatz wieder richtig besetzt, doch darunter eine Lücke entstanden. Wie schließen wir diese? Na einfach ganz genauso. Jeder Teil des Heaps ist ja wieder ein kleiner Heap für sich!

Wurzel entfernen	7 ist größtes Kind, hinaufziehen	jetzt ist 6 maximal	der Knoten hat ohnehin nur ein Kind	der Heap ist wieder maximal

Beobachtungen:

- wir entfernen L[0] und schieben den Rest des Baumes hoch. Dabei bleibt immer eine Lücke, die langsam nach unten purzelt ('percolation'=durchsickern).
- schließlich bleibt der letzte Platz L[9] leer. Beim Entfernen des nun ganz oben sitzenden größten Elementes (des zweitgrößten der gesamten Liste) bleiben dann die letzten 2 Plätze frei, usw.
- diese leeren Plätze können wir sofort verwenden, um die aktuell aufgefundenen Maximalwerte zu speichern. Allerdings erst NACH der Reparatur des Heaps.

Es geht sogar noch klüger:

Wozu mit einem unbesetzten Platz arbeiten, der nach unten rutscht? Setzen wir doch einfach einen kleinen Zahlenwert ein, dessen Platz dann ohnehin unten ist. Gut – aber welchen Wert? Na einfach den letzten in der noch unsortierten Liste, und wir können den entfernten Maximalwert sofort richtig in die Gesamtliste einbauen! Allerdings müssen wir dazu einen Zähler mitführen, der angibt, wie viele Elemente noch zum Heap gehören, die dahinter stehenden sind dann bereits sortiert. Diese Vorgehensweise ist aber auch notwendig, da es sonst passieren könnte, dass mitten im Heap eine freie Lücke bleibt. So können wir die vorschriftsmäßige Füllung garantieren.

1. Unsere Sortierfunktion für den Heap L[0]...L[n] arbeitet also so:
 L[0] ist das maximale Element.
 - vertausche L[0] mit L[n]
 - n = n-1
 jetzt sitzt das maximale Element HINTER dem neuen (defekten) Heap, ein zu kleines Element ist an der Wurzel, der Heap ist um einen Platz verkürzt worden.
2. um den defekten Heap L[0] bis L[n] zu reparieren mach folgendes:
 Beginne ganz oben.
 Ist der Wert dieses Knotens nicht größer als der seiner beiden Kinder, so tausche ihn mit dem größeren Kind aus.
 Und mach das solange, bis der Wert passt, oder der Wert an eine Stelle durchgesickert ist, die keinen Kindknoten mehr hat.

Im folgenden Beispiel sind die Werte, die nicht mehr zum Heap gehören, vom Baum getrennt und grau unterlegt.

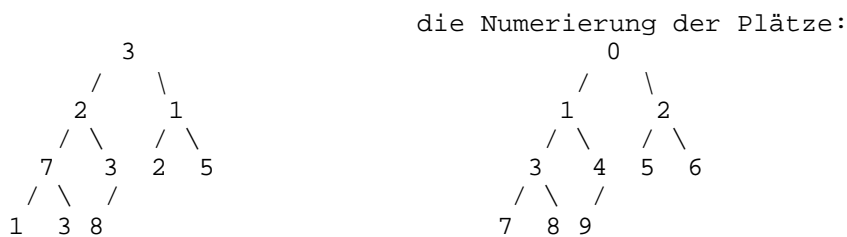
[9, 7, 6, 4, 6, 5, 3, 2, 3, 1]	[1, 7, 6, 4, 6, 5, 3, 2, 3, 9]	[7, 1, 6, 4, 6, 5, 3, 2, 3, 9]	[7, 6, 6, 4, 1, 5, 3, 2, 3, 9]	[3, 6, 6, 4, 1, 5, 3, 2, 7, 9]
ERSTER DURCHGANG: tausche erstes und letztes Element, verkürze den Heap.	1 sickert zum größeren Kind – sie tauschen Plätze	wiederum sickert 1 hinunter und hat schon sein Ziel erreicht.	Reparatur fertig. NÄCHSTER DURCHGANG: tausche erstes und letztes Element, verkürze den Heap.	3 sickert hinunter. Es ist egal ob links oder rechts. Wir nehmen rechts.

[6, 6, 3, 4, 1, 5, 3, 2, 7, 9]	[6, 6, 5, 4, 1, 3, 3, 2, 7, 9]	[2, 6, 5, 4, 1, 3, 3, 6, 7, 9]	[6, 2, 5, 4, 1, 3, 3, 6, 7, 9]	[6, 4, 5, 2, 1, 3, 3, 6, 7, 9]
der Wert des Kindes ist 5 – also tauschen und die Reparatur ist fertig.	NÄCHSTER DURCHGANG: tausche erstes und letztes Element, verkürze den Heap.	2 sickert nach links	und nochmals.	und wieder ist die Reparatur geglückt. der NÄCHSTE DURCHGANG kann beginnen.

Wir sehen: WENN wir einen Max-Heap vorliegen hätten, könnten wir die Liste sortieren. Aber wie bauen wir einen solchen Heap auf ????

Wir haben doch schon erkannt, dass ein Heap aus lauter Teilen besteht, die selbst wieder die Heap-Eigenschaften erfüllen. Diese Erkenntnis drehen wir nun um: wir stellen sicher, dass alle kleinen Heaps korrekt sind und 'reparieren' einfach nach oben hin durch.

Beispiel: verwandle [3,2,1,7,3,2,5,1,3,8]. in einen korrekten Heap.



Die Liste besteht anfangs aus den unsortierten Werten L[0] bis L[9]. Die zweite Hälfte der Liste sind einsame 'Blätter' des Baumes ohne Kinder – hier gibt es nichts zu verbessern. Dann führen wir Reparaturen durch: zuerst ab Stelle 4, dann Stelle 3, usw. bis Stelle 0.

Wir müssen unserer Reparatur-Funktion nur die Nummer des Knotens übergeben können, von dem abwärts gearbeitet werden soll, und die Sache ist gelaufen!!!

9 Elemente, wir beginnen bei Knoten 4 . Die Reparatur tauscht 8 und 3	Knoten 3 ist in Ordnung	Knoten 2: die 1 sickert nach unten	Knoten 1: der Zweier sickert nach unten	Knoten 0: der Dreier wird gegen den Siebener getauscht, fertig.

Die Sortierung läuft also so ab:

- 1.) Erzeuge den Heap, indem Du ihn einfach füllst und dann von unten nach oben reparierst
- 2.) Hole immer wieder das maximale Element ab (Wurzel des Baums) und repariere den Heap

vorerst ein paar praktische Hilfsroutinen:

<pre>def fullrange(a,b,step=1): if step>0: return range(a,b+1,step) else: return range(a,b-1,step) def swap(L,a,b): L[a],L[b] = L[b],L[a]</pre>	<p>range() inklusive Randwerte</p> <p>vertausche zwei Listenelemente</p>
<pre>def links(n): return 2*n+1 def rechts(n): return 2*n+2</pre>	<p>linker Kindknoten</p> <p>rechter Kindknoten</p>

<pre>def repariere(L,pos,letzt): l = links(pos) r = rechts(pos) if (l<=letzt) and (L[l]>L[pos]): maximal = l else: maximal = pos if (r<=letzt) and (L[r]>L[maximal]): maximal = r if maximal != pos: swap(L,maximal,pos) repariere(L,maximal,letzt)</pre>	<ul style="list-style-type: none"> - Liste L, ab Knoten pos, Nummer letztes Heapelement - linkes Kind - rechtes Kind - es gibt das linke Kind und es ist größer - dieses ist also größer - sonst... - bleibt der Ausgangsknoten maximal - rechtes Kind existiert und ist größer - maximal darauf setzen - falls ein Kind größer war - Wert hinunter sicken lassen - und von dort aus weiter reparieren
<pre>def erzeugeHeap(L): letzte = len(L)-1 for i in fullrange(len(L)/2,0,-1): repariere(L,i,letzte)</pre>	<p>verwandelt L in einen MaximalHeap</p> <ul style="list-style-type: none"> - Nummer des letzten Elementes - von der Mitte nach oben - Heapeigenschaft erzeugen
<pre>def heapsort(L): erzeugeHeap(L) for letzte in fullrange(len(L)-1,1,-1): swap(L,0,letzte) repariere(L,0,letzte-1) return L</pre>	<ul style="list-style-type: none"> - baue den Heap auf - letztes Heaposition von hinten nach vorne - Wurzel nach unten tauschen - repariere mit abgehängtem letzten Element - gib die sortierte Liste zurück

Aufgaben:

- 1.) Wie lange braucht Heapsort für 500,1000,2000,4000 Elemente?
- 2.) Die doppelte Anzahl von Werten braucht etwa mal so viel Zeit.
- 3.) Heapsort ist von Geschwindigkeitsordnung $n \log(n)$

Heapsort ist also ein 'schneller' Algorithmus.

Er ist üblicherweise etwas langsamer als Quicksort, doch in **jedem** Fall von Ordnung $n \log n$, während Quicksort im schlechtesten Fall zu einer 'langsamen' Methode der Ordnung n^2 wird. Details erfährst Du, wenn wir Quicksort besprechen.