Warum Bottom-up Heapsort?

Die Idee hinter bottom-up Heapsort ist, die Feststellung von eben aufzugreifen und die Konstante bei $n \log n$ kleiner zu bekommen.

Bottom-up Heapsort funktioniert ganz genau so wie Standard-Heapsort, nur die reheap-Prozedur wird anders implementiert. Am einfachsten stellt man sich das so vor, dass das einsinkende Element zunächst bis zu einem Blatt sinkt, unabhängig von seinem eigenen Wert. Dazu benötigt man in jedem Schritt nur einen Vergleich!

Anschließend steigt das eingesunkene Element so lange auf, bis das über ihm liegende Element kleiner als dieses (oder gleich diesem) ist. Auch hier genügt ein Vergleich pro Schritt.

Analyse von Bottom-up Heapsort

Da beim Absinken jeweils nur ein Vergleich nötig ist (nämlich der Vergleich $a[2i] < a[2i+1]$?), werden für das Absinken aller n Elemente insgesamt nicht mehr als $n \log n$ Vergleichsoperationen benötigt.
Aber wieviel kostet das anschließende Aufsteigen?

Wir benutzen noch einmal das Ergebnis über die unendliche Summe der Terme $i/2^i$.

Offensichtlich werden weniger als die Hälfte aller n Elemente um eine Ebene aufsteigen, weniger als ein Viertel um zwei, und so weiter – also weniger als $n/2^i$ um i Ebenen.

Die Gesamtzahl der Aufstiege ist also kleiner als n mal die unendliche Summe der $i/2^i$, und damit in $O(n)$.

Analyse (Fortsetzung)

Auf der vorherigen Folie haben wir ein bisschen *geschummelt*...

Die Behauptung, die Zahl der um i Ebenen steigenden Elemente wäre nicht größer als $n/2^i$, ist zwar im Durchschnitt plausibel.

Aber im worst case kann es durchaus sein, dass viele Elemente weit nach oben steigen, um dann wieder von anderen verdrängt und weiter nach unten geschoben zu werden!

Daher braucht man eine genauere Untersuchung, um die Komplexität von Bottom-up Heapsort im worst case bzw. im average case zu analysieren.

Die folgenden Ergebnisse geben wir hier ohne Beweis an:

Bottom-up Heapsort benötigt $1.5n \log n + o(n \log n)$ Vergleiche im worst case, bzw. $n \log n + o(n \log n)$ im average case.

Ultimatives Heapsort

Wir machen zunächst ein Gedankenspiel:

Wenn wir zur Sortierung von n Elementen $2n$ Speicherplätze verwenden könnten, würden wir zu den Elementen $a[1, \ldots, n]$ noch weitere n Elemente mit Wert ∞ legen und dann Heapsort durchführen bis alle n Elemente einsortiert sind.

Welchen Aufwand hätten wir dann?

Wir brauchen $O(n)$ Vergleiche für den Heapaufbau.

Aber dann genügen $n \log n$ Vergleiche für das Heraussortieren der Elemente $a[1]$ bis $a[n]$ an die n hinzugefügten Plätze.

Warum nur $n \log n$, also $1 \cdot \log n$ pro Element?

> Weil die einsinkenden Elemente immer den Wert ∞ haben, d.h. wir lassen sie wie bei Bottom-up Heapsort ganz nach unten rutschen, aber ohne Aufstieg!

Plan für Ultimate-Heapsort

Wir wollen aber ein In-place Sortierverfahren verwenden, d.h. den zusätzlichen Speicherplatz haben wir nicht.

Was können wir tun, um den Gedanken der vorigen Folie dennoch weiterzuverfolgen?

Wir müssen dafür sorgen, dass die erste Hälfte des Eingabe-Arrays die kleinen, und die zweite Hälfte die großen Elemente enthält, d.h.

$$i < \lceil n/2 \rceil, j \geq \lceil n/2 \rceil \quad \Longrightarrow \quad a[i] < a[j]$$

Wenn wir das geschafft haben, können wir die $a[j]$ mit $j \geq \lceil n/2 \rceil$ behandeln als wenn sie den Wert $a[j] = \infty$ hätten. Damit hätten wir mit Aufwand $\frac{n}{2} \log n$ die Hälfte des Arrays sortiert und müssten nur noch einen rekursiven Aufruf auf der halben Größe machen, um den Rest zu sortieren. Die Details folgen auf den nächsten Folien.

Die ersten Aktionen

Wir müssen die Bedingung erfüllen, dass alle Elemente der ersten Array-Hälfte kleiner als alle Elementen der zweiten Hälfte sind. (OBdA seien alle Elemente verschieden!)

Dazu berechnen wir zuerst den Median der Elemente $a[1, \ldots, n]$.
Die Medianberechnung ist in Linearzeit möglich - das werden wir nachher zeigen.

Nun benutzen wir den Median als Pivot-Element und machen wie bei Quicksort eine Aufteilung in zwei Teile, wobei im ersten Teil alle Elemente liegen werden, die kleiner oder gleich dem Median sind (also genau die Hälfte!), und im rechten Teil die, die größer als der Median sind.

Jetzt liegt die gewünschte Ausgangsposition vor. Wir können in den ersten $n/2$ Elementen zunächst einen Heap aufbauen und diese Elemente dann wie bei Standardheapsort eines nach dem anderen „herauspflücken“ und an das Ende des Arrays sortiert einfügen.