

Teil 2: Sortieren
Allgemeine Sortierverfahren

BOTTOM-UP HEAPSORT

Verbesserung von HEAPSORT

HEAPSORT erweist sich in der Praxis als sehr schnelles Sortierverfahren. Sein Verhalten ist sowohl im schlechtesten Fall als auch im Mittel sehr gut.

Allerdings werden wir noch sehen, dass es im Mittel von **QUICKSORT** geschlagen wird.

Frage

Kann HEAPSORT verbessert werden?

Dazu betrachten wir die Prozedur **Heapify**(S, j, r).

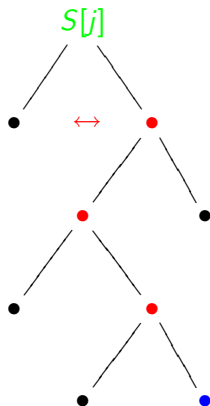
Der potentielle Einsinkpfad

In jedem Schritt wird der **größere** der beiden Nachkommen gewählt.

Ist das einzusickernde Element kleiner, wird es mit seinem Nachkommen vertauscht und das Verfahren eine Ebene tiefer fortgesetzt.

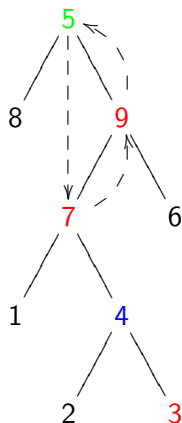
Durch diese Vorgehen ergibt sich ein **potentieller Einsinkpfad** von Position j aus, indem man beginnend an Position j den Pfad der größeren Nachkommen geht.

Auf diese Art erhält man ein **spezielles Blatt** des Heaps.



Carlson's Idee

- 1 Erstelle den potentiellen Einsinkpfad.
- 2 Suche mittels binärer Suche entlang des potentiellen Einsinkpfades die Austauschstelle.
 - ▶ Da nur $S[j]$ die Heap-Bedingung in dem Teilbaum verletzt, bilden die Schlüssel auf dem Einsinkpfad eine monoton fallende Folge (Nachkommen kleiner als Vorgänger)
 - ▶ Es muss also nur die Stelle im Einsinkpfad gefunden werden, bei der zum „ersten Mal“ der Schlüssel kleiner ist als der einzusickernde Wert.
- 3 Schiebe die Schlüssel oberhalb der gefundenen Position auf dem Pfad um eine Position nach oben und kopiere den einzusickernden Schlüssel an die frei gewordene Position.



Carlson's Idee

k sei die Tiefe der untersten Schicht und i die Tiefe des einzusickernden Knotens. Dann benötigt die Erstellung des Einsinkpfades $k - i$ Vergleiche (einen pro Schicht) und die binäre Suche nach der korrekten Position $\log(k - i) \leq \log \log n$ Vergleiche.

Damit ergibt sich die Gesamtzahl der benötigten Vergleiche als:

$$V_{CHS}(n) = n \log n + \Theta(n \log \log n).$$

Für $n \geq 10^6$ haben Experimente gezeigt, dass dieses Verfahren schneller ist als alle QUICKSORT Varianten.

Frage

Warum ist diese HEAPSORT Variante wesentlich schneller?

Das Einsinken der Schlüssel

Die Antwort auf diese Frage liegt in der Wegnahmephase.

In jeder Runde wird ein Wert in der untersten Ebene mit dem Maximum vertauscht. Anschließend sickert dieser Wert mittels Heapify ein.

Da vor dem Tausch ein Max-Heap vorliegt, sind die Werte in der untersten Ebene wahrscheinlich eher **klein**.

Damit muss ein relativ kleiner Wert von der Wurzel weit nach unten sickern.

Etwas genauer: In den untersten Ebenen befinden sich mehr als die Hälfte der Schlüssel.

- unterste Ebene / Blätter: ca. $\frac{n}{2}$ Schlüssel
- vorletzte Ebene: ca. $\frac{n}{4}$ Schlüssel
- drittletzte Ebene: ca. $\frac{n}{8}$ Schlüssel

D.h. ca. $\frac{3}{4}n$ in den beiden untersten Ebenen (bei vollständigem Heap).

Das Einsinken der Schlüssel

Konsequenz

Die Wahrscheinlichkeit, dass das einzusickernde Element zu den $\frac{3}{4}n$ kleinsten Elementen gehört ist relativ groß.

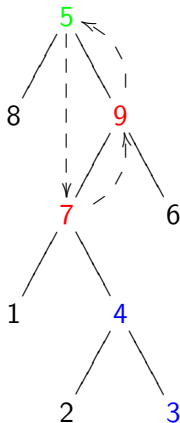
Damit ist es sehr wahrscheinlich, dass das Element bis in die untersten Ebenen sickern muss.

Das legt nahe, dass es günstig ist den Einsickerungspfad von **unten nach oben** statt von **oben nach unten** nach der korrekten Position zu durchsuchen.

Genau dies tut **BOTTOM-UP HEAPSORT**.

BOTTOM-UP HEAPSORT

- 1 Erstelle den potentiellen Einsinkpfad. ($k - i$)
Vergleiche (von Ebene i bis Ebene k)
- 2 Suche beginnend im Blatt des Pfades die Austauschstelle, d.h. gehe von unten nach oben, bis der Vorgänger größer oder gleich dem einzusickernden Schlüssel ist.
- 3 Schiebe die Schlüssel oberhalb der gefundenen Position auf dem Pfad um eine Position nach oben und kopiere den einzusickernden Schlüssel an die frei gewordene Position.



BOTTOM-UP HEAPSORT - Die Analyse

Satz

BOTTOM-UP HEAPSORT benötigt

- 1 *im schlechtesten Fall höchstens $1.5n \log n + O(n)$ Vergleiche. [Wegener 90]*
- 2 *im Mittel $n \log n + O(n)$ Vergleiche. [Schaeffer/Sedgewick 93]*

Obwohl BUHS (BOTTOM-UP HEAPSORT) schlechter aussieht als das von Carlson vorgeschlagene Verfahren (mit binärer Suche), ist es im Mittel dennoch besser.

BOTTOM-UP HEAPSORT - Die Analyse

Beweis (Worst-Case BUHS)

F sei eine konkrete zu sortierende Folge von Schlüsseln.

d_k sei die Einsinktiefe von HEAPSORT (HS) nach der k -ten Wegnahme.
D.h. in der k -ten Runde werden $2d_k$ Vergleiche ausgeführt. Damit gilt

$$HS_W(F) \leq 2 \sum_{k=1}^n d_k =: 2D.$$

m_k sei sowohl bei HS, als auch bei BUHS die Tiefe des Heaps nach der k -ten Wegnahme.

Da der Heap in jeder Runde um ein Element verkleinert wird, gilt

$$M := \sum_{k=1}^n m_k = \sum_{i=1}^n \lfloor \log i \rfloor.$$

Beweis (Worst-Case BUHS (Fortsetzung))

Für die Heap Tiefen $m_k k$ gilt:

$$M = \sum_{k=1}^n m_k = \sum_{i=1}^n \lfloor \log i \rfloor = \lfloor \log n \rfloor n - O(n).$$

Zum Beweis

Damit ist die Höchstzahl der von BUHS durchgeführten Vergleiche gegeben als:

$$BUHS_W(F) \leq \underbrace{\sum_{k=1}^n m_k}_{\text{Einsinkpfad}} + \underbrace{\sum_{k=1}^n (m_k - d_k)}_{\text{Suche von Unten}} = 2 \sum_{k=1}^n m_k - \sum_{k=1}^n d_k = 2M - D$$

Beweis (Worst-Case BUHS (Fortsetzung))

Damit gilt:

$$BUHS_W(F) \leq 2M - D \leq 2M - \frac{1}{2}HS_W(F) \leq 2n \log n - \frac{1}{2}HS_W(F).$$

Wir werden später sehen, dass

$$HS_W(F) \geq n \log n - O(n)$$

gilt (allgemeine untere Schranke für Sortierverfahren).

Damit ergibt sich

$$BUHS_W(F) \leq 1.5n \log n + O(n),$$

was die obere Schranke für den schlechtesten Fall beweist.



Beweis (Average-Case BUHS)

Für das Verhalten im Mittel haben Schaeffer und Sedgewick gezeigt, dass

$$HS^{avg}(n) \geq 2n \log n - O(n \log \log n)$$

gilt (Beweis folgt). Damit folgt für das mittlere Verhalten von BUHS:

$$\begin{aligned} BUHS_W^{avg}(n) &\leq 2n \log n - O(n) - \frac{1}{2} HS_W^{avg}(n) \\ &\leq 2n \log n - O(n) - n \log n + O(n \log \log n) \\ &= n \log n + O(n \log \log n). \end{aligned}$$



Lemma (Sedgewick/Schaeffer)

$$HS^{avg}(n) \geq 2n \log n - O(n \log \log n).$$

Beweis (Average-Case BUHS (Fortsetzung))

In den beiden Phasen von HEAPSORT wird durch die Funktion Heapify die anfängliche Reihenfolge der Schlüssel verändert. Für die Analyse im Mittel wirkt sich dabei insbesondere die Aufbauphase negativ aus, da nicht klar ist, wie die Reihenfolgen nach dem Aufbau des Heaps verteilt sind.

Um diese Schwierigkeit zu umgehen, werden wir betrachten, wieviele Bits notwendig sind, um die Vorgänge während eines Ablaufs von Heapify so zu kodieren, dass wir sie wieder rückgängig machen können.

Auf diese Art werden wir eine Bitfolge einer bestimmten Länge erhalten, die uns eindeutig ermöglicht die Eingabereihenfolge aus der sortierten Folge zu rekonstruieren.

Beweis (Average-Case BUHS (Fortsetzung))

Über den Vergleich dieser Größe mit $n!$, der Anzahl der möglichen Eingaben, werden wir dann eine Schätzung für die mittlere Anzahl von Vergleichen pro Aufruf von Heapify erhalten.

Um einen Aufruf von Heapify rückgängig machen zu können, müssen wir den Einsinkpfad der Wurzel kodieren.

Informal geht dies in der Form

links - rechts - links - links -Ende.

*Verwendet man **0 für links** und **1 für rechts** ergibt sich bei Einsinktiefe d eine Bitsequenz $b_1 \dots b_d$. Da Heapify aber mehrmals aufgerufen wird, muss auch d mit kodiert werden, damit man das ganze Verfahren durch eine Bitsequenz beschreibt.*

Insgesamt ergibt sich die Frage, wie man eine einzelne Sequenz $b_1 \dots b_d$ von Bits kodiert, so dass man ihr Ende immer noch erkennen kann, wenn man viele solcher Kodierungen hintereinander schreibt.

Beweis (Average-Case BUHS (Fortsetzung))

Ein einfacher Ansatz ist es hinter jedes Bit eine zusätzliche 0 zu schreiben und am Ende eine 1, d.h. die Kodierung $c(b_1 \dots b_d)$ hätte die Form

$$c(b_1 \dots b_d) := b_1 0 b_2 0 \dots b_d 1.$$

dadurch würde sich die Länge der Sequenz verdoppeln.

Effizienter ist eine Kodierung in der folgenden Form

$$\bar{c}(b_1 \dots b_d) = c(\text{bin}(d)) b_1 \dots b_d,$$

wobei $\text{bin}(d)$ die Binärkodierung von d ist. D.h. wir kodieren erst die Länge der Sequenz auf die oben beschriebene Art und schreiben anschließend einfach die Bitsequenz auf.

Diese Kodierung hat somit die Länge

$$|\bar{c}(b_1 \dots b_d)| = d + 2 \lfloor \log d \rfloor.$$

Beweis (Average-Case BUHS (Fortsetzung))

Wir verwenden allerdings eine etwas andere Kodierung, nämlich:

$$\bar{c}(b_1 \dots b_d) = c(\text{bin}(\log n - d))b_1 \dots b_d,$$

mit der Länge

$$|\bar{c}(b_1 \dots b_d)| = d + 2\lfloor \log(\log(n) - d) \rfloor.$$

Wir nehmen nun an, dass wir eine solche Kodierung bei jedem aufruf von Heapify mitführen.

Sind $d_1, \dots, d_{\lfloor \frac{1}{2}n \rfloor + n}$ die Einsinktiefen, so ergibt sich insgesamt eine Bitsequenz der Länge

$$\sum_{i=1}^{\lfloor \frac{3n}{2} \rfloor} (d_i + 2\lfloor \log(\log(n) - d_i) \rfloor).$$

Beweis (Average-Case BUHS (Fortsetzung))

Man kann sich leicht überzeugen, dass dieser Wert maximal wird, wenn alle d_i den gleichen Wert

$$d := \frac{\sum_{i=1}^{\lfloor \frac{3n}{2} \rfloor} 2d_i}{3n}$$

annehmen. D.h. die Länge der Bitsequenz lässt sich nach oben abschätzen durch

$$\frac{3}{2}nd + \frac{3}{2}2n \lfloor \log(\log(n) - d) \rfloor.$$

Wir betrachten nun, welche Länge die Bitsequenzen haben müssen, damit wir nur $2^{-n}n!$ viele verschiedene Eingabesequenzen kodieren können.

Beweis (Average-Case BUHS (Fortsetzung))

Mittels der *Stirlingschen Formel*

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

erhält man die Abschätzung

$$\log\left(\frac{n!}{2^n}\right) = \log(n!) - n \geq n \log n - 3n,$$

d.h. höchstens ein Anteil von 2^{-n} vielen Permutationen kann mit weniger als $n \log n - 3n$ Bits kodiert werden.

Für die restlichen $(1 - 2^{-n})n!$ Permutationen werden mehr Bits benötigt, d.h. für diese gilt

$$\frac{3}{2}nd + 3n \lfloor \log(\log(n) - d) \rfloor \geq n \log n - 3n.$$

Beweis (Average-Case BUHS (Fortsetzung))

$$\frac{3}{2}nd + 3n\lfloor \log(\log(n) - d) \rfloor \geq n \log n - 3n$$

und somit

$$d \geq \frac{2}{3} \log n - 2 - 2\lfloor \log(\log(n) - d) \rfloor \quad (1)$$

Für $d \geq 0$ folgt durch Einsetzen in die rechte Seite von (1):

$$d \geq \frac{2}{3} \log n - 2 - 2\lfloor \log \log n \rfloor = \frac{2}{3} \log n - O(\log \log n). \quad (2)$$

Damit ergibt sich

$$\sum_{k=1}^{\lfloor \frac{3n}{2} \rfloor} d_k = \frac{3}{2}nd = n \log n - O(n \log \log n).$$



HEAPSORT und BOTTOM-UP HEAPSORT im Vergleich

	HEAPSORT	BOTTOM-UP HEAPSORT
Worst Case	$2n \log(n+1) - O(n)$	$1.5n \log n + O(n)$
Im Mittel	$\sim 2n \log n + O(n \log \log n)$	$n \log n + O(n)$