Inhaltsverzeichnis

1 Vorwort **1**

2 Einleitung **6**

3 Der WEAK-HEAPSORT Algorithmus **9**
3.1 Der Weak-Heap als binärer Baum 9
3.2 Einbettung in ein 1-dimensionales Array 15
3.2.1 Die Funktion Gparent 16
3.2.2 Die Funktion Merge 16
3.2.3 Die Funktion WeakHeapify 17
3.2.4 Die Funktion MergeForest 20
3.2.5 Die Funktion WeakHeapSort 21

4 Verbesserungen von WEAK-HEAPSORT **24**
4.1 Entschärfung des worst-case 24
4.2 Best-case bei linearem Zusatzspeicher 26

5 Die best-case Analyse von WEAK-HEAPSORT **29**

6 Die worst-case Analysen **43**
6.1 Die worst-case Analyse von HEAPSORT 43
6.2 Die worst-case Analyse von BOTTOM–UP–HEAPSORT 45
6.3 Die worst-case Analyse von MDR–HEAPSORT 51
6.4 Die worst-case Analyse von WEAK-HEAPSORT 54

7 Die Anzahl von Heaps und Weak-Heaps **59**
7.1 Die Anzahl von Heaps 59
7.2 Die Anzahl von Weak-Heaps 62

8 Die Rückwärtsanalyse **67**
8.1 Die Aufbauphase von HEAPSORT 69
8.2 Die Aufbauphase von WEAK-HEAPSORT 70
8.3 Die Auswahlphase von HEAPSORT 77
8.4 Die Auswahlphase von WEAK-HEAPSORT 79

9 Die Verteilung auf der Menge der Weak-Heaps **83**

10 Die average-case Analysen der Aufbauphase **90**
10.1 Aufbauphase von HEAPSORT 90
10.2 Aufbauphase von BOTTOM-UP-HEAPSORT 97

11 Die average-case Analysen der Auswahlphase **98**
11.1 Die average-case Analyse von BOTTOM-UP-HEAPSORT 99
11.2 Die average-case Analyse von WEAK-HEAPSORT 107
11.3 Rückwärtige Generierung aller Weak-Heaps 120

12 Eine Datenstruktur für diverse Operationen **123**
12.1 Das MIN-MAX Problem 123
12.2 Konvertierung eines Heaps 125
12.3 Der Weak-Heap als Priority Queue 127

13 Die Sortieralgorithmen in der Praxis **129**

A Eulers Summationsformel **138**

B Abbildungs– und Tabellenverzeichnis **140**

C Literaturverzeichnis **142**

1 Vorwort

Die Ordnung ist die Lust der Vernunft, aber die Unordnung ist die Wonne der Phantasie.

Paul Claudel

Jeder Mensch, der in einem Lexikon geblättert, eine Telefonnummer gesucht, Dateiinhaltsverzeichnisse gelesen oder Karten gespielt hat, wird die Übersichtlichkeit von und die Effizienz des Umganges mit geordnet vorliegenden Objekten bestätigen können.
Die Vor- bzw. Aufbereitung von miteinander vergleichbaren Objekten, wird als Ordnen, Auslesen, Gruppieren oder auch Sortieren bezeichnet.
Die Objekte werden auch Datensätze genannt und lassen sich in die für die Sortierung relevanten Schlüssel und die übrige Information einteilen. Je nach Aufgabenstellung kann diese Einteilung variieren. Die Schlüssel können sehr komplex, die Operationen zwischen ihnen somit sehr aufwendig sein.
Da die Datenmengen häufig groß sind und die Sortierung nach der Festlegung einer Vorgehensweise eine mechanische Aufgabe ist, werden Computer insbesondere als Informationssysteme für die Verarbeitung von elektronisch erfaßten Daten eingesetzt.
Der Entwurf von guten Sortieralgorithmen ist so zu einem fundamentalen Problem in der Informatik geworden. Der besondere Reiz der bekannten Verfahren liegt besonders in der Einfachheit der Ideen, die gut erlern-, vermittel- und übertragbar sind. Desweiteren ermöglichen sie vielfach eine präzise und beispielhafte Analyse der Effizienz. Demnach werden Sortieralgorithmen in jeder programmiertechnischen Ausbildung bis hin zur Universität gelehrt.
Die Zeitkomplexität (vergl. Wegener(1994)) des Sortierproblems ist mittlerweile sowohl im schlechtesten als auch im mittleren Fall bekannt, d.h. zu den theoretisch ermittelten unteren Grenzen existieren Algorithmen, die die gleiche Wachstumsordnung besitzen.
Somit stellt sich die Frage, warum noch an der Weiterentwicklung und Analyse von Sortierverfahren geforscht wird. Auf der Suche nach Gründen finden sich folgende Punkte:

1. Jede Verbesserung eines Sortierverfahrens oder der eventuell zugrunde liegenden Datenstruktur führt direkt zur Verbesserung unzähliger Programme, die einen Zugriff auf sortierte Daten benötigen.

2. Je nach intern oder extern vorliegenden Daten, komplexen oder einfachen Schlüsseln, großen oder kleinen Datenmengen, Grad der Vorsortiertheit bzw. Verteilung der Datensätze, Anforderungen an die Stabilität etc. unterscheiden sich die zu empfehlenden Algorithmen enorm.

3. Sei es die worst-case Analyse von SHELLSORT oder die average-case Analyse von HEAPSORT, viele Probleme bedürfen noch einer genauen

Klärung.

4. Die Angabe der Effizienz eines Algorithmus in der Lanudau'schen O–Notation verbirgt die Vorfaktoren der zu beschreibenden Funktion. Um das unterschiedliche Laufzeitverhalten von Sortieralgorithmen zu erklären, muß die Güte der Approximation verbessert werden.

5. Fortschritte des sequentiellen Sortieren haben eine enorme Durchschlagskraft in der Fachwelt, die sich seit ihren Anfängen mit dem Problem beschäftigt und über eine gemeinsame Grundlage verfügt.

In dieser Diplomarbeit wird der Algorithmus WEAK–HEAPSORT vorgestellt und analysiert, der von Ronald D. Dutton 1993 in der Zeitschrift BIT veröffentlicht wurde.
Duttons Ausführungen ist zu wenig Aufmerksamkeit geschenkt worden.
Die wichtigsten Merkmale dieses Algorithmus führen den Leser an die ausgearbeiten Abschnitte:

Sortieren durch Auswählen In dieser HEAPSORT-Variante besteht der Sortiervorgang von n Elementen aus einem Aufbau einer *PriorityQueue*–Datenstruktur (biggest in, first out) namens Weak-Heap mit anschließendem n-maligen Entnehmen eines Elementes (vergl. Kapitel 3).

Aufbau Die Weak-Heap Datenstruktur kann mit optimalen $n-1$ Vergleichen und $O(n)$ übrigen Operationen aufgebaut werden (vergl. Kapitel 3.2.3) oder ohne Vergleiche aus einem Heap gewonnen werden (vergl. Kapitel 12.2). Das MIN-MAX-Problem kann mit Weak-Heaps optimal gelöst werden (vergl. Kapitel 12.1).

Einfachheit der Idee Da sich die Heap-Anforderung als zu restriktiv erweist, wird sie abgeschwächt (vergl. Kapitel 3.1).

Vergleichskriterien Der WEAK-HEAPSORT-Algorithmus erfüllt alle 7 Kriterien an einen Sortieralgorithmus (vergl. Kapitel 2).

Performance Die Laufzeitbestimmungen des best- und worst-cases sind einfach: Mit $k = \lceil \log n \rceil$ benötigt der Algorithmus mindestens $nk - 2^k + 1 \geq n \log n - n$ und höchstens $nk - 2^k + n - k \leq (n-1) \log n + 0.1n$ Vergleiche (vergl. Kapitel 3.2). Insbesondere sind die ermittelten Grenzen *scharf* (vergl. Kapitel 5 und 6.4), d.h. es gibt Beispiele, bei denen diese Grenzen auch angenommen werden. Die empirisch ermittelten praktischen Vergleiche zu bekannten Sortierverfahren bestärken die außerordentliche Geschwindigkeit des Algorithmus (vergl. Kapitel 13).

Verbesserung Der worst-case läßt sich mindestens um den Summanden $\log n$ entschärfen (vergl. Kapitel 4).

Binärer Baum Die generierten und als binäre Bäume aufgefaßten Weak-Heaps lassen sich in einen Allgemeinen Heap einbetten, sind untereinander gleichverteilt, und die Anzahl ist damit leicht zu bestimmen (vergl. Kapitel 3.1 und 7.2).

Rückwärtsbetrachtung Der Algorithmus läßt sich sowohl in der Aufbau- als auch Auswahlphase rückwärtig studieren und ermöglicht so, Aussagen über die Struktur, Anzahl und Verteilung von Weak-Heaps zu machen (vergl. Kapitel 8 und 9). Diese Betrachtung bietet vielversprechende Ansätze zur average-case Analyse (vergl. Kapitel 11.2). Letztendlich können alle Weak-Heaps auch rückwärtig aus dem einzigen Weak-Heap der Größe 1 generiert werden (vergl. Kapitel 11.3).

Datenstruktur Der Weak-Heap bietet als *PriorityQueue* auch im allgemeinen Fall effiziente Operationen an. Das Einfügen ist in ungefähr $(\log n)/2$, das Löschen in exakt $\lceil \log(n+1) \rceil - 1$ Schritten möglich (vergl. Kapitel 12.3).

Den Stärken des Algorithmus steht eine zu diskutierende Schwäche entgegen. Pro Schlüssel ist ein Zusatz- bzw. Informationsbit für den Algorithmus unabdingbar.

Da allgemeine Schlüsselvergleichsverfahren erst für das Sortieren komplexer Schlüssel (reelle Zahlen, viele Entititäten) zu empfehlen sind, wird die Zeit für den Vergleich als auch den Tausch zweier Objekte als groß angesehen. Der Tausch von zwei Datensätzen bzw. deren Schlüssel kann jedoch durch den Tausch von Zeigern auf die entsprechenden Stellen simuliert werden. So ist die Anzahl der Vergleiche in der Theorie zum Maßstab der Analyse geworden: Ein Zeiger und damit insbesondere ein Bit sind demnach klein gegenüber der Schlüsselgröße.

Vielfach wird der volle zu Verfügung stehende Wertebereich der auftretenden Schlüssel nicht ausgeschöpft und das Zusatzbit kann häufig effektiv, z.B. als Vorzeichenbit, integriert und verarbeitet werden. Praktisch wird somit kein zusätzlicher Speicherplatz verbraucht.

Auch der zur Sortierung am häufigsten verwendete QUICKSORT Algorithmus benötigt Zusatzspeicher in Form eines Rekursionsstacks mindestens logarithmischer Größe, indem die lokalen Indexvariablen und die Rücksprungadressen abgelegt werden müssen. Die meisten Implementierungen benötigen jedoch aus zwei Gründen einen Stack linearer Größe: aus einer naheliegenden – jedoch ungeschickten – Abarbeitung der aufgeteilten Objektmenge (der kleinere Teil muß vor dem größeren abgearbeitet werden), als auch durch die Unfähigkeit vieler gängiger Interpreter bzw. Compiler, End- bzw. tail-Rekursionen zu erkennen (vergl. Abelson und Sussman (1984)).

Die Anforderung, innerhalb der gegebenen Datenarraystruktur ohne großen Speicheraufwand sortieren zu können, erscheint trotz der Notwendigkeit von linear vielen Zusatzbits erfüllt.

Demnach sollte der WEAK-HEAPSORT Algorithmus sowohl in programmiertechnischen Ausbildungen als auch an der Universität neben den anderen Verfahren gleichberechtigt gelehrt werden.
Literaturverweise zu Sätzen oder Hilfssätzen befinden sich durchgehend hinter dem Wort 'Beweis'. Wesentliche Veränderungen zu den referierten Originalarbeiten werden durch Fußnoten hervorgehoben.
In erster Linie danke ich Herrn Univ. Prof. Dr. Ingo Wegener für die ausgezeichnete fachliche und persönliche Betreuung meiner Diplomarbeit. Weiterhin gilt mein Dank Erik Dörnenburg und Frank Rettberg, die mich bei der Implementierung der Algorithmen für das in Kapitel 13 beschriebene Programm *Sorting in Action* unterstützten. Erik Dörnenburg und Armin M. Warda haben mir außerdem einige wertvolle Hinweise bei den abschließenden Korrekturen der Arbeit gegeben.
Ich möchte diese Diplomarbeit Dr. John Kelly widmen, dessen sprachfußnotenreiche Vorlesungen am University College Dublin ich während meines Auslandaufenthaltes genießen durfte und der Anfang dieses Jahres verstarb. Trotz oder gerade aufgrund der von mir geteilten mathematischen Leidenschaft beschäftigte er sich mit der Grenze menschlicher und künstlicher Intelligenz (Kelly (1993)), dessen Essenz ich gerne abschließend sinngemäß zitieren möchte und somit eine Absage der Übertragung von Vergleichbarkeiten menschlichen und maschinellen Sortierens auf die der Intelligenz erteile:

1. Die Natur menschlicher Intelligenz ist schwer erfaßbar und läßt keine fügsame Formalisierung oder effektive Charakterisierung zu.

2. Soweit heutzutage verstanden und mittels konzeptioneller Analyse enthüllt, existiert eine große Divergenz zwischen der Ontologie (Lehre des Seins) der Maschine und des Menschen.

3. Computer werden irrtümlicherweise als Symbolprozessoren charakterisiert. In ihnen selbst sind sie bloß Signalprozessoren.

4. Computer–Systeme, sowohl die traditionelle Hardware oder Software oder die modernen neuronalen Netze, können am besten als Texte verstanden werden.

5. Computer haben keine Möglichkeit, in einem tiefen Sinn des Wortes zu *handeln*. Sie *tun* nichts.

6. Falsche Konzepte über die Möglichkeiten von Computern und Fehlinterpretationen der Computerrechenleistung entstehen aus:

   - übertriebener Vermenschlichung von Maschinen,
   - dem Zauber eines existentiellen Irrtums oder dem Trugschluß falschplazierter Korrektheit,

- der Abhängigkeit von einer einfachen Liste, die eine notwendige Einheit darbieten soll,
- unangemessene Einstellung zur Stärke der Wissenschaft[...],
- dem Glauben an die Zulässigkeit von Sprache.

7. Die Abhängigkeit der Computer von der menschlichen Sprache zeigt die Grenzen ihrer Möglichkeiten auf, da menschliche Erkenntnis, sogar sprachverbundene Erkenntnis, ultimativ in Unaussprechlichkeit wurzelt.

Stefan Edelkamp
Universität Dortmund
September 1995

2 Einleitung

Die Wissenschaft fängt eigentlich erst da an, interessant zu werden, wo sie aufhört.

Justus von Liebig

Allgemeine Schlüsselvergleichsverfahren benötigen im worst-case nach Stirlings Approximation von $n!$ (vergleiche Anhang) durch Betrachtung des binären Entscheidungsbaumes mindestens

$$\lceil \log(n!) \rceil = n \log n - n \log e + \Theta(\log n) \approx n \log n - 1.4427n$$

und im Mittel mindestens

$$\lceil \log(n!) \rceil - \frac{1}{n!} 2^{\lceil \log(n!) \rceil} \geq \lceil \log(n!) \rceil - 1$$

Vergleiche.
Um Sortierverfahren untereinander vergleichen zu können, werden 7 Kriterien aufgestellt (vergl. Wegener (1995)):

1. Das Sortierverfahren sollte allgemein sein. Objekte aus beliebig geordneten Mengen sollten sortiert werden können.

2. Die Implementation des Sortierverfahrens soll einfach sein.

3. Das Sortierverfahren soll internes Sortieren ermöglichen. Dabei steht neben den Arrayplätzen nur sehr wenig Platz zur Verfügung.

4. Die durchschnittliche Zahl von wesentlichen Vergleichen, d.h. Vergleichen zwischen zu sortierenden Objekten soll klein sein. Sie soll für ein möglichst kleines c durch $n \log n + cn$ beschränkt sein.

5. Die worst-case Zahl von wesentlichen Vergleichen soll klein sein. Sie soll für ein möglichst kleines c' durch $n \log n + c'n$ beschränkt sein.

6. Die Zahl der übrigen Operationen wie Vertauschungen, Zuweisungen und unwesentliche Vergleiche soll höchstens um ein konstanten Faktor größer als die Zahl der wesentlichen Vergleiche sein.

7. Die CPU-Zeit für jede übrige Operation soll klein gegenüber der CPU-Zeit für einen Vergleich zweier komplexer Objekte sein.

BUCKETSORT basiert auf der Darstellung der zu sortierenden Elemente und ist somit kein allgemeines Verfahren (Kriterium 1).

MERGESORT ist ein Divide-and-Conquer Algorithmus und benötigt für $n = 2^k$ durch ein paralleles Durchlaufen der schon sortierten Teillisten nur $n \log n - n + 1$ Vergleiche, allerdings ein Array der Länge $2n$. Da n recht groß sein kann, ist man an 'In-Situ' Algorithmen (Kriterium 3) interessiert. Daher ist MERGESORT nur zum externen Sortieren geeignet.

INSERTIONSORT (Steinhaus (1958)) ist eines der einfachsten Sortierverfahren. Es benötigt durch n-maliges Einfügen mittels binärer Suche weniger als

$$\sum_{i=1}^{n-1} \lceil \log(i+1) \rceil = \sum_{i=2}^{n} \lceil \log(i) \rceil \leq \sum_{i=2}^{n} \log(i+1) = \log(n!) + n - 1$$

Vergleiche, aber die Anzahl der Vertauschungen liegt selbst im average-case in $\Theta(n^2)$. Das Interesse gilt aber Sortieralgorithmen, deren Anzahl von Vertauschungen und anderen nicht-essentiellen Operationen klein ist (Kriterium 6).

SHELLSORT (Shell (1959)) besitzt als weiteren Parameter eine Folge von natürlichen Zahlen $h_1, \ldots, h_t$. Dabei wird sukzessiv INSERTIONSORT auf den Elementen aufgerufen, die einen Abstand h_i mit $i = t, \ldots, 1$ zueinander besitzen. Die geeignete Auswahl der Abstandsfolge ist sehr entscheidend für das Laufzeitverhalten von SHELLSORT. Es werden im folgenden die Ergebnisse für den worst-case von Operationen (Vergleiche und Vertauschungen) zusammengefaßt. Shell (1959) schlug die Abstandsfolge $(1, 2, \ldots, 2^k)$ vor, doch liefert sie im Falle $n = 2^k$ eine quadratische Laufzeit. Eine von Hibbard (1963) vorgeschlagene Sequenz ergab in der Analyse von Papernov und Stasevich (1965) $O(n^{3/2})$. Pratt (1979) gab eine $\Theta(\log^2 n)$ lange Folge an, die zu $\Theta(n \log^2 n)$ vielen Operationen führt. Sedgewick (1986) verbesserte die Grenze von $O(n^{3/2})$ auch für $O(\log n)$ beschränkte Folgen auf $O(n^{4/3})$ und in Zusammenarbeit mit Incerpi (1985) gab er als Weiterentwicklung dieser Idee die obere Grenze von $O(n^{1+\epsilon/\sqrt{\log n}})$ für ein $\epsilon > 0$ an. Poonen (1993) zeigte, daß dies die bestmögliche Schranke ist. Cypher (1993) bewies für SHELLSORT Netzwerke und für monoton fallende Abstandsfolgen eine Mindestgröße von $\Omega(n \log^2 n / \log\log n)$. Letztendlich fand Poonen (1993) eine untere Grenze von $\Omega(n(\log n / \log\log n)^2)$ Operationen unabhängig von den gewählten Abstände auch für SHELLSORT als allgemeines Sortierverfahren. Die Analysen begründen sich vielfach auf die Anzahl der Inversionen (vergl. Knuth (1973)) und auf die Betrachtung des Frobenius-Problems (Brauer (1942)).
Festgehalten werden sollte, daß SHELLSORT für vorsortierte oder mittelgroße Eingaben (wenige Tausend) schneller ist als nahezu alle bekannten schnellen Sortierverfahren.

QUICKSORT (Hoare (1962)) ist das wohl bekannteste, auf einer Partitionierung der Schlüsselmenge beruhende Sortierverfahren. Damit liegt der wesentliche Aufwand des Divide-and-Conquer Ansatzes im divide–Schritt ($n - 1$

Vergleiche). Dort wird die endgültige Position k eines ausgewählten Elementes x ermittelt und die Schlüsselmenge in einen von x dominierten bzw. in einen x dominierenden Teil eingeteilt.
Der Algorithmus benötigt bei ungünstiger Aufteilung der Objektmenge $\Theta(n^2)$ viele Vergleiche. Für den average-case an Vergleichen $V(n)$ findet sich durch die Mittelung der möglichen Fälle folgende Rekursionsgleichung:

$$V(n) = \begin{cases} 0 & n \in \{0,1\} \\ n-1+\frac{1}{n}\sum_{k=1}^{n}(V(k-1)+V(n-k)) & n \geq 2 \end{cases}$$

Diese Summe läßt sich zu folgender Gleichung vereinfachen:

$$V(n) = 2(n+1)H_n - 4n,$$

wobei $H_n = \sum_{i=1}^{n} 1/i$ die n-te Partialsumme der Harmonischen Reihe bezeichnet. Die Harmonische Reihe läßt sich gut approximieren (vergl. Anhang) und man erhält für die mittlere Anzahl der Vergleiche:

$$V(n) \approx 1.386 n \log n - 2.846 n + O(\log n)$$

CLEVER-QUICKSORT ist eine von Hoare (1962) vorgeschlagene verbesserte Variante von QUICKSORT. Da das Partitionselement weit vom Median entfernt sein kann, wird das mittlere Element x von drei zur Partition vorgeschlagenen Objekten ermittelt, und die Partitionierung der übrigen Objekte bezüglich dieses Elementes durchgeführt. Bei ungünstiger Aufteilung der Objektmenge sind wieder $\Theta(n^2)$ viele Vergleiche nötig, doch der average-case wird deutlich gesenkt.
Der Median dreier Objekte kann in durchschnittlich 8/3 Vergleichen gefunden werden, was leicht durch Fallunterscheidung zu verifizieren ist. Damit benötigt der divide–Schritt im Mittel $n-3+8/3 = n-1/3$ Vergleiche. Die Wahrscheinlichkeit, daß x an der Position k steht, ist gleich $(k-1)(n-k)/\binom{n}{3}$, da es dann $k-1$ Positionen für das kleinere und $n-k$ Positionen für das größere Objekt gibt.
Für den average-case an Vergleichen $V(n)$ gilt demnach folgende Rekursonsgleichung:

$$V(n) = \begin{cases} 0 & n \in \{0,1\} \\ 1 & n = 2 \\ n - \frac{1}{3} + \binom{n}{3}^{-1} \sum\limits_{k=1}^{n} (k-1)(n-k)(V(k-1)+V(n-k)) & n \geq 2 \end{cases}$$

Diese Summe läßt sich für $n \geq 6$ zu folgender Gleichung zusammenfassen (Wegener (1995)):

$$\begin{aligned} V(n) &= \frac{12}{7}(n+1)H_{n-1} - \frac{477}{147}n + \frac{223}{147} + \frac{252}{147n} \\ &\approx 1.188 n \log n - 2.255 n + O(\log n). \end{aligned}$$

Wird der Median nicht aus drei sondern aus $2k+1$ Elementen gebildet, läßt sich das mittlere Laufzeitverhalten noch verbessern und es ergibt sich eine Komplexität von $\alpha_k n \log n + O(n)$ mit konstanten $\alpha_k > 1$. Bei wachsendem k konvergiert α_k gegen 1 (vergl. van Emden (1970)).
Derzeit ist keine QUICKSORT-Variante bekannt, deren average-case durch $n \log n + o(n \log n)$ beschränkt ist.

HEAPSORT wird in Kapitel 6.1 beschrieben.

BOTTOM-UP-HEAPSORT wird in Kapitel 6.2 beschrieben.

MDR-HEAPSORT wird in Kapitel 6.3 beschrieben.

3 Der WEAK-HEAPSORT Algorithmus

Ordnung ist die Verbindung des vielen nach einer Regel.

Immanuel Kant

Analog zum HEAPSORT ist auch beim WEAK-HEAPSORT für das Verständnis die Betrachtung des Heaps als binärer Baum von seiner Einbettung in ein Array zu trennen.
In diesem Kapitel wird der WEAK-HEAPSORT Algorithmus (Dutton (1992) und (1993)) vorgestellt und das Laufzeitverhalten analysiert.

3.1 Der Weak-Heap als binärer Baum

Wenn sich ein Baum zu beugen versteht, wird er nie vom Wind gebrochen.

Aus Afrika

Im folgenden bezeichne T einen binären Baum, $a[i]$, $1 \leq i \leq n$, den Wert des Schlüssels für den Knoten i, $root(T)$ die Wurzel von T, $rT(x)$ (bzw. $lT(x)$) den rechten (bzw. linken) Teilbaum von x, dessen Wurzel $rchild(x)$ ($lchild(x)$) ist, umgekehrt ist $Parent(x)$ der Vorgänger im Baum. Mit $|T|$ wird die Anzahl der Knoten in T beschrieben. Sei $depth(x)$ die Tiefe von x, d.h. der Abstand zur Wurzel, und $Depth(T)$ das Maximum über alle $x \in T$. Desweiteren bezeichne $height(x)$ die Höhe von x im Baum, d.h. den maximalen Abstand von x zu einem Blatt, $c(x) \in \{0, 1, 2\}$ die Anzahl der Kinder von x.

Definition 1 *Ein (Max-)***Weak-Heap** *ist ein binärer Baum T mit den folgenden Eigenschaften:*

(W1) *Der Wert an jedem Knoten ist größer oder gleich den Werten an den Knoten des rechten Teilbaumes, d.h.* $\forall x \in T, \forall y \in rT(x) : a[x] \geq a[y]$.

(W2) *Die Wurzel hat keinen linken Teilbaum, d.h* $lT(root(T)) = \emptyset$.

(W3) *Außer der Wurzel haben die Knoten nur dann weniger als zwei Kinder, wenn sie auf den letzten beiden Leveln des Baumes liegen, d.h.* $c(x) < 2 \Rightarrow ((x = root(T)) \vee (depth(x) \geq Depth(T) - 1))$.

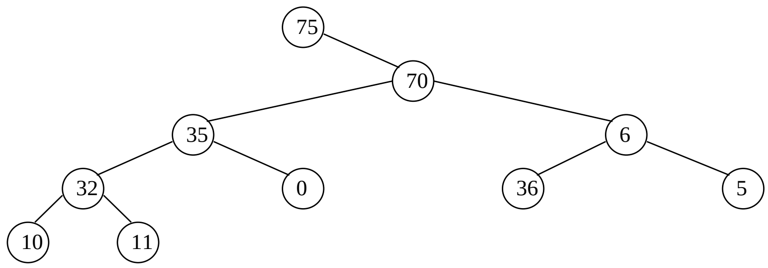

Abbildung 1: Beispiel eines Weak-Heaps.

Damit ist gesichert, daß trotz der geringeren Anforderungen der maximale Wert am Wurzelknoten anliegt. Die Definition wirkt sich unmittelbar auf die sich ergebenen Teilstrukturen aus (vergl. Abb. 2).

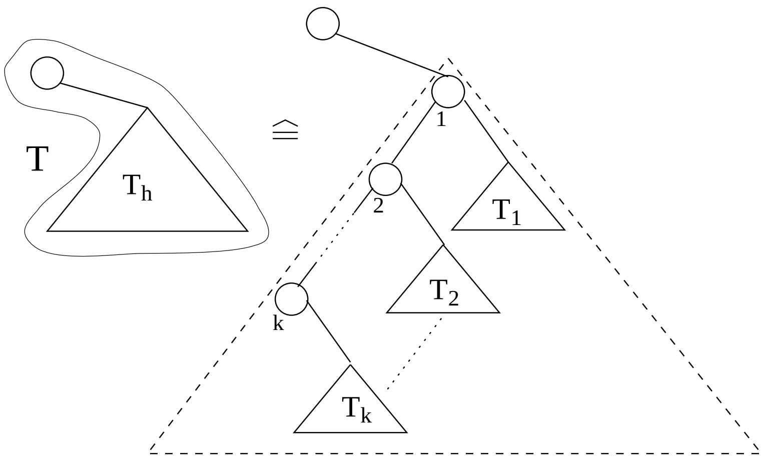

Abbildung 2: Schematische Darstellung der Weak-Heap-Datenstruktur.

Definiert man mit $\langle i,j\rangle$ den binären Baum in Weak-Heap-Form aus Wurzel i und rechtem Teilbaum mit Wurzel j, so ist für alle $i \in \{1,\ldots,k\}$ $\langle i, root(T_i)\rangle$ ein Weak-Heap, denn (W3) vererbt sich an die jeweilige Teilstruktur. Über die Größe der in Abb. 2 dargestellen Teilbäume T_i, $i \in \{1,\ldots,k\}$, läßt sich folgendes Resultat erzielen:

Hilfssatz 1 *Für alle $i \in \{1,\ldots,k\}$ gilt:*

$$2^{k-i} - 1 \leq |T_i| \leq 2^{k-i+1} - 1.$$

Beweis: Ein vollständiger binärer Baum der Tiefe k hat $2^{k+1}-1$ Knoten. Nach (W3) hat T_i minimal die Größe eines vollständigen binären Baumes der Tiefe $k-i-1$ und maximal die Größe eines vollständigen binären Baumes der Tiefe $k-i$. □

Bei der Sortierung mittels eines Weak-Heaps wird in Analogie zu HEAPSORT das Wurzelelement als Maximum in die schon sortierte Reihenfolge übernommen und die Strukturinvarianz der Definition wieder hergestellt. Der Weak-Heap zerfällt in einen Wald von mehreren Teil-Weak-Heaps, die wieder miteinander verbunden werden müssen. Um die Beziehung zwischen den Wurzeln des sich ergebenen Waldes und der ehemaligen Wurzel zu beschreiben, betrachte man folgende *großelterliche* Relation.

Definition 2

$$Gparent(x) = \begin{cases} Gparent(Parent(x)) & \textit{falls } x \textit{ linkes Kind,} \\ Parent(x) & \textit{falls } x \textit{ rechtes Kind.} \end{cases}$$

$$y \in S_x \Longleftrightarrow Gparent(y) = x.$$

Damit ist
$S_x = \{rchild(x)\} \cup \{y \mid y \text{ ist von } rchild(x) \text{ nur über linke Kinder erreichbar}\}$.

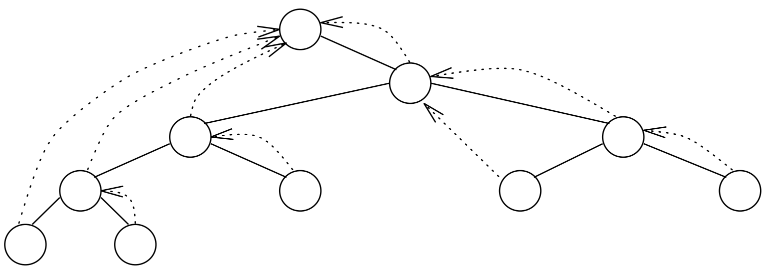

Abbildung 3: Veranschaulichung der Relation *Gparent*.

So ist z.B. in Abb.2 $S_{root(T)} = \{1, 2, \ldots, k\}$. Seien $\langle x, rchild(x)\rangle$ und $\langle y, rchild(y)\rangle$ zwei gegebene Weak-Heaps, $rchild(x) = root(T_1)$, $rchild(y) = root(T_2)$. Ohne Beschränkung der Allgemeinheit (o.B.d.A.) wird angenommen, daß $a[x] \geq a[y]$ ist. Dann ergibt die folgende $Merge_1(\langle x, rchild(x)\rangle, \langle y, rchild(y)\rangle)$ – Sequenz:

1. $root(T') = x$,
2. $lchild(y) = rchild(x)$,
3. $rchild(root(T')) = y$,

einen Baum mit (W1) und (W2), allerdings nicht notwendigerweise auch mit (W3) (vergl. Abb. 4).

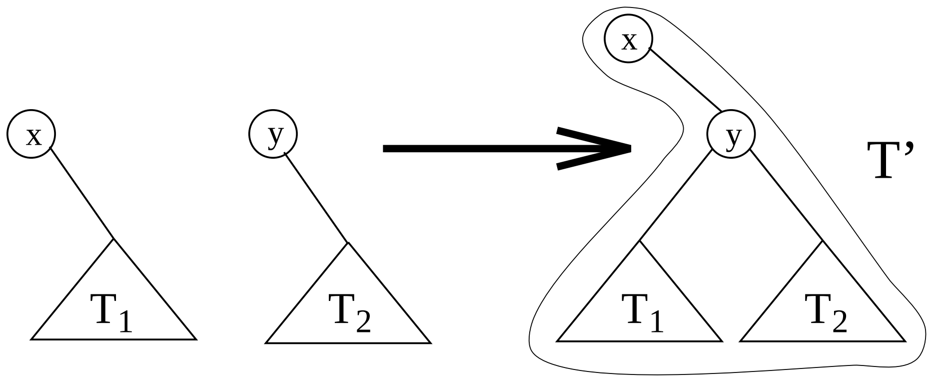

Abbildung 4: Merge von zwei Weak-Heaps, o.B.d.A. $a[x] \geq a[y]$.

Um diese Bedingung zu sichern, wird der Begriff der Kompatibilität zwischen zwei Weak-Heaps eingeführt:

Definition 3 *Zwei Weak-Heaps T', T'' werden kompatibel genannt, wenn die Tiefen aller Knoten mit weniger als zwei Kindern mit Ausnahme der Wurzeln sich maximal um 1 unterscheiden, d.h.*

$$\forall x \in \{z \in T' - root(T') \mid c(z) < 2\}, \forall y \in \{z \in T'' - root(T'') \mid c(z) < 2\} : \\ \mid depth(x) - depth(y) \mid \leq 1.$$

Nun kann ein modifiziertes Merge[1] eingeführt werden, daß auf einem Knoten x und einem Baum T wirkt. Sei $y = root(T)$, unter der Voraussetzung, daß für alle $z \in lT(y)$ $a[x] > a[z]$ gilt, definiere:

$$Merge_2(x, T) := Merge_1(\langle x, root(lT(y))\rangle, \langle y, root(rT(y))\rangle). \qquad (1)$$

Es ergibt sich folgende Fallunterscheidung:

[1] In Duttons Arbeit findet sich eine solche für das Verständnis nützliche Unterscheidung der $Merge$–Operationen nicht.

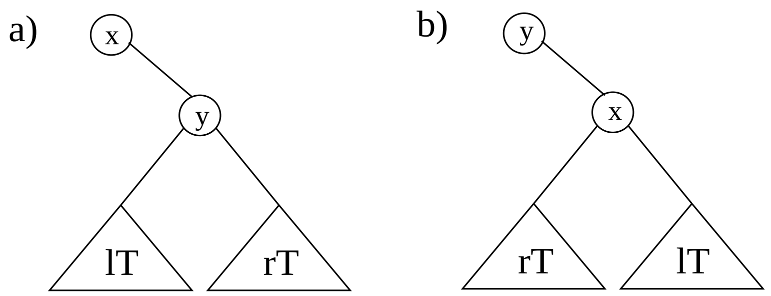

Abbildung 5: Modifiziertes Merge: a) $a[x] \geq a[y]$, b) $a[y] > a[x]$.

a) $a[x] \geq a[y]$: x wird zur Wurzel des entstehenden Weak-Heaps, wobei $rchild(x)$ auf y gesetzt wird.

b) $a[y] > a[x]$: y wird zur Wurzel des entstehenden Weak-Heaps, wobei $lchild(x)$ auf $rchild(y)$, $rchild(x)$ auf $lchild(y)$ und letztendlich $rchild(y)$ auf x gesetzt wird.

Satz 1 *Sei T binärer Baum mit $y = root(T)$, sowie $\langle x, lT(y)\rangle, \langle y, rT(y)\rangle$ kompatible Weak-Heaps. Dann ist $Merge_2(x,T)$ ein Weak-Heap.*

Beweis: (Dutton (1992)[2]) Da $\langle x, lT(y)\rangle$ ein Weak-Heap ist, folgt für alle $k \in lT(y) : a[x] \geq a[k]$, ebenso ist $\langle y, rT(y)\rangle$ ein Weak-Heap und damit gilt für alle $k \in rT(y) : a[y] \geq a[k]$. Für den Fall $a[x] \geq a[y]$ ergibt sich mittels Transitivität für alle $k \in rT(y) : a[x] \geq a[k]$ und damit insgesamt für alle $k \in T : a[x] \geq a[k]$. Analog ergibt sich im Falle $a[y] > a[x]$ für alle $k \in lT(y) : a[y] \geq a[k]$ und damit für alle $k \in T : a[y] > a[k]$. Die Vertauschung der Teilbäume in T ist notwendig, da keine Dominanz von $a[x]$ gegenüber Schlüsseln in $rT(y)$ gesichert ist. Damit sind (W1) und (W2) erfüllt und die Kompatibilität sichert in beiden Fällen unmittelbar (W3) des so entstehenden Weak-Heaps. □

Die Operation $Merge_2(x,T)$ benötigt nur einen Schlüsselvergleich, allerdings muß die Vertauschung der Teilbäume realisiert werden.
Zur Initialisierung des WeakHeaps T ($WeakHeapify(T)$) wird jeder Knoten j mittels $Merge_2$ mit seinem Großelternteil $Gparent(j)$ verbunden. Dabei müssen die $n-1$ Aufrufe von den Blättern zur Wurzel (bottom–up) hin organisiert werden, d.h. $Merge_2(Gparent(j), j)$ kann nur dann aufgerufen werden, wenn sowohl $Merge_2(Gparent(rchild(j)), rchild(j))$ als auch $Merge_2(Gparent(lchild(j)), lchild(j))$ terminiert sind. Damit kann gesichert

[2]Ebendortiges (Ebd.) Lemma 2.1 und Theorem 3.2 wurden zu diesem Satz konzeptionell neu verbunden.

werden, daß immer größer werdende Weak-Heaps gebildet werden, so daß letztendlich gilt:

Satz 2 *Nach der Terminierung von $WeakHeapify(T)$ ist T ein Weak-Heap.*

Beweis: (Dutton (1992)[3]) Annahme: Es existiert ein j, so daß $Merge_2(Gparent(j), j) = \langle Gparent(j), j\rangle$ kein Weak-Heap ist. Dann wähle ein solches j maximal in dem Sinne, daß der rechte Teilbaum entweder leer oder die Weak-Heap-Rückgabe $\langle Gparent(rchild(j)), rchild(j)\rangle = \langle j, rchild(j)\rangle$ eines vorangegangenen $Merge_2-$Aufrufes ist und der linke Teilbaum entweder leer oder der Weak-Heap $\langle Gparent(lchild(j)), lchild(j)\rangle = \langle Gparent(j), lchild(j)\rangle$ ist. Die Ausführung von $Merge_2(Gparent(j), j)$ liefert aufgrund Satz 1 nun einen Widerspruch zur Annahme. Also gilt für alle j, daß $\langle Gparent(j), j\rangle$ ein Weak-Heap ist, somit insbesondere $T = \langle root(T), rchild(root(T))\rangle$. □

Um die Neuverbindung des sich durch die Wurzelentnahme ergebenden Waldes aus Teil-Weak-Heaps mittels $MergeForest$ zu organisieren, sei $S_{root}(T) = \{1, 2, \ldots, k\}$ gegeben und die Wurzel entfernt. Aufbauend auf $Merge_2$ kann dieser so entstehende Pfad bottom–up genutzt werden. Sei dazu m ein beliebig(!) gewähltes Element auf dem untersten Level, das die nächste Wurzelposition repäsentieren wird. In dem Falle, daß k kein rechtes Kind hat, erweist sich die Wahl von $m = k$ als sehr vorteilhaft, da sich dadurch der Pfad auf $\{1, 2, \ldots, k-1\}$ verkürzt, d.h. x kann mit $k-1$ statt mit k initialisiert werden. Nacheinander führe fogendes Anweisungspaar bis zum Erreichen der Wurzel aus (vergl. Abb.6):

1. $Merge_2(m, x)$,
2. $x \leftarrow Parent(x)$.

Hierbei repräsentiere x jeweils die Teilbäume T_i zu $x = root(T_i)$.
Die Korrektheit dieses Ansatzes belegt folgender

Satz 3 *Sei $\langle i, T_i\rangle, i \in \{1, \ldots k\}$, ein Weak-Heap-Wald. Nach Ausführung der Prozedur MergeForest für ein beliebig gewähltes m ist $\langle m, 1\rangle$ ein Weak-Heap.*

Beweis:(Dutton (1992)[4]) Nach der Initialisierung (Fall $m \neq k$) ist x der weitestlinks stehende Knoten ohne Kinder, mit dem m verbunden wird. Da x Blatt ist, sind die Voraussetzungen für die $Merge_2$- Operation trivialerweise erfüllt. Somit ist $\langle m, x\rangle$ ein Weak-Heap. Aufgrund der einleitenden Bemerkung ist für alle $i \in \{1, \ldots, k\}$: $\langle i, root(T_i)\rangle$ ein Weak-Heap, d.h. in unserem Falle ist nach $x \leftarrow Parent(x)$ $\langle x, root(T_x)\rangle$ ein Weak-Heap. Wieder sind alle Voraussetzungen für $Merge_2$ erfüllt, was letztendlich darin gipfelt, daß $\langle m, 1\rangle$ ein Weak-Heap wird. Im Falle $m = k$ ist x größer als $rchild(x)$, also auch ein Weak-Heap, und die Beweisführung erfolgt analog. □

[3]Ebd. Lemma 2.3 und Theorem 3.3 wurden zu diesem Satz konzeptionell neu verbunden.
[4]Ebd. Lemma 2.2 und Theorem 3.4 wurden zu diesem Satz konzeptionell neu verbunden.

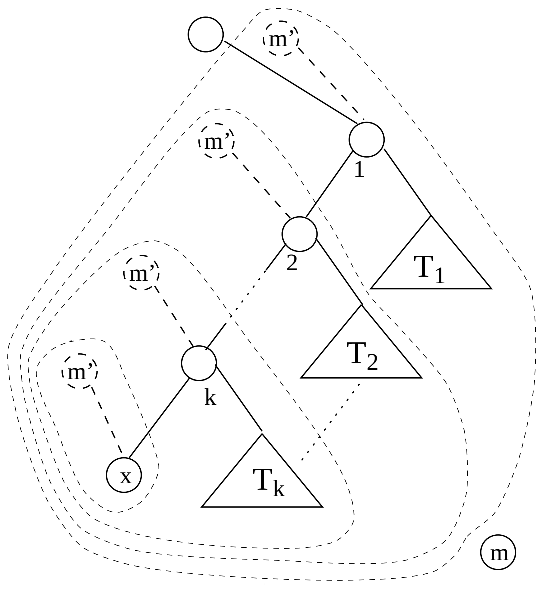

Abbildung 6: Sukzessive Vermengung von m mit den eweiligen Teilbäumen bei $MergeForest$.

3.2 Einbettung in ein 1-dimensionales Array

Wenn wir nicht von vorne anfangen, dürfen wir nicht hoffen, weiter zu kommen.

Johann Gottfried Seume

Die Einbettung eines binären Baumes T der Größe n in ein Array $a[1], \ldots, a[2^n - 1]$ ist eine wohldefinierte injektive Abbildung mit den folgenden Eigenschaften:

1. Die Wurzel $root(T)$ wird auf die Arrayposition 1 abgebildet.
2. Falls die $v \in T$ auf die Arrayposition x abgebildet wird, so wird $lchild(v)$ auf die Position $2x$ und $rchild(v)$ auf die Position $2x + 1$ abgebildet.

Ist der Binärbaum von links aufgefüllt, so genügen die n Arraypositionen $a[1], \ldots, a[n]$ für dessen Einbettung, d.h. die Zuordnung ist bijektiv. Gilt zusätzlich für alle $1 \leq \lfloor j/2 \rfloor < j \leq n$: $a[\lfloor j/2 \rfloor] \leq a[j]$, so spricht man von einem Heap. Der Vorteil der Arraydarstellung liegt in der konstanten Zugriffszeit für die gegebenen Positionen.

Um auch Weak-Heaps in einem Array der Größe n einbetten und effizient verwalten zu können, muß die Vertauschung von Teilbäumen in konstanter Zeit realisiert werden. Die Idee ist nun, ein Boole'sches Array $Reverse$ zu verwalten, dessen Einträge $Reverse[x] \in \{0, 1\}$ angeben, ob die zu einem Knoten

x korrespondierenden Teilbäume vertauscht sind oder nicht. Das rechte Kind von x kann mit $2x + Reverse[x]$, das linke Kind mit $2x + 1 - Reverse[x]$ geroutet werden. Dementsprechend wird die von der gewünschten Belegung von *Reverse* abhängige Einbettung eines Weak-Heaps T der Größe n in ein Array $a[0], \dots, a[n-1]$ wie folgt festgelegt:

1. Die Wurzel $root(T)$ wird auf die Arrayposition 0 abgebildet.

2. Falls der Knoten $v \in T$ auf die Arrayposition x abgebildet wird, so wird $lchild(v)$ auf die Position $2x + Reverse[x]$ und $rchild(v)$ auf die Position $2x + 1 - Reverse[x]$ abgebildet.

Sind alle *Reverse*–Bits auf 0 gesetzt, so entspricht diese Abbildung mit Ausnahme der Wurzelstelle der Standardeinbettung eines von links aufgefüllten binären Baumes. Die einen Binärbaum in Weak-Heap Struktur beschreibende Eingabe des Algorithmus wird zum Beipiel so in ein Array eingebettet werden.
Wichtig ist die Beobachtung, daß nicht alle binären Bäume, die (W1)-(W3) erfüllen, sich auf ein Array der Größe n abbilden lassen (vergl. Kapitel 4.2 und 7.2).
Es können und werden jedoch in dieser Datenstruktur alle angegebenen und auf der Eingabe aufbauenden konzeptionellen Algorithmen verwirklicht und analysiert.

3.2.1 Die Funktion Gparent

Da x genau dann linkes Kind ist, wenn $odd(x) = Reverse[\lfloor x/2 \rfloor]$, realisiert folgende Funktion definitionsgemäß Gparent:

```
PROCEDURE Gparent(j)
  WHILE odd(j) = Reverse[j DIV 2] DO j = j DIV 2
  RETURN j DIV 2
END Gparent.
```

Die Prozedur *Gparent* wird nur in der Aufbauphase des Sortiervorganges aufgerufen, wo für alle betrachteten Werte j `Reverse[j DIV 2]` noch mit 0 initialisiert ist. Darum läßt sich das Prädikat `odd(j)=Reverse[j DIV 2]` auch durch das Prädikat `even(j)` ersetzen.

3.2.2 Die Funktion Merge

Die Übertragung von $Merge_2$ wird von nun an verkürzt mit $Merge$ bezeichnet und gestaltet sich wie folgt:

```
PROCEDURE Merge(i,j)
  IF a[i] < a[j] THEN
    swap(a[i],a[j]);
```

```
    Reverse[j] = 1 - Reverse[j];
  FI
END Merge.
```

3.2.3 Die Funktion WeakHeapify

Weak-Heaps können in der Arrayeinbettung wie folgt generiert werden:

```
PROCEDURE WeakHeapify
  FOR j = n-1 DOWNTO 1 DO Merge(Gparent(j),j)
END WeakHeapify.
```

Nach Satz 2 berechnet diese Funktion einen initialen Weak-Heap, da $Merge(Gparent(j), j)$ nur dann aufgerufen wird, wenn sowohl $Merge(Gparent(rchild(j)), rchild(j))$ als auch $Merge(Gparent(lchild(j)), lchild(j))$ terminiert sind.

Hilfssatz 2

$$\sum_{k=0}^{n} k2^k = (n-1)2^{n+1} + 2 \tag{2}$$

Beweis: Der Beweis stützt sich auf die Finitisimalrechnung (vergl. Graham (1989)): Sei $\Delta f(x) = f(x+1) - f(x)$ die Definition eines Differenzoperators in diskreter Analogie zum Differentialoperator im kontinuierlichen Fall. Dann gilt (vergl. mit Fundamentalsatz der Analysis):

$$\sum g(x)\delta x = f(x) + C :\Longleftrightarrow g(x) = \Delta f(x),$$

wobei $\sum g(x)\delta x$ die uneigentliche Summe von g bezeichnet, d.h eine Klasse von Funktionen auf die der Differenzoperator wieder $g(x)$ liefert. C repräsentiert eine Funktion p mit $p(x+1) = p(x), x \in Z$. Über diesen Ansatz lassen sich die eigentlichen Summen bestimmen:

$$\begin{aligned} \textstyle\sum_a^b g(x)\delta x &= f(x)\,\big|_a^b \\ &= f(b) - f(a). \end{aligned}$$

Wichtig ist letztendlich die folgende induktiv beweisbare Betrachtung:

$$\sum_{k=a}^{b-1} g(k) = \textstyle\sum_a^b g(x)\delta x.$$

Die Gleichsetzung $f(x) = \Delta f(x) = f(x+1) - f(x)$ liefert eine Lösung $f(x) = 2^x$. Außerdem ist $\Delta x = x + 1 - x = 1$.

Analog zur partiellen Integration existiert eine partielle Summationsformel:

$$\sum u(x)\Delta v(x)\delta x = u(x)v(x) - \sum \Delta u(x)v(x+1)$$

Für den zu beweisenden Spezialfall gilt demnach:

$$\begin{aligned}\sum_{k=0}^{n} k2^k &= \sum\nolimits_0^{n+1} x2^x \delta x \\ &= \left(x2^x - 2^{x+1}\right)\big|_0^{n+1} \\ &= ((n+1)2^{n+1} - 2^{n+2}) - (0 \cdot 2^0 - 2^1) \\ &= (n-1)2^{n+1} + 2.\square\end{aligned}$$

Satz 4 *Für $n = 2^k$ beträgt die Gesamtzahl $BO(n)$ von Bitoperationen von Weak–Heapify:*

$$BO(n) = 2n - k - 2.$$

und somit ist $BO(n) \in O(n)$.

Beweis: Abb. 7 veranschaulicht die formelmäßige Berechnung von $BO(n)$:

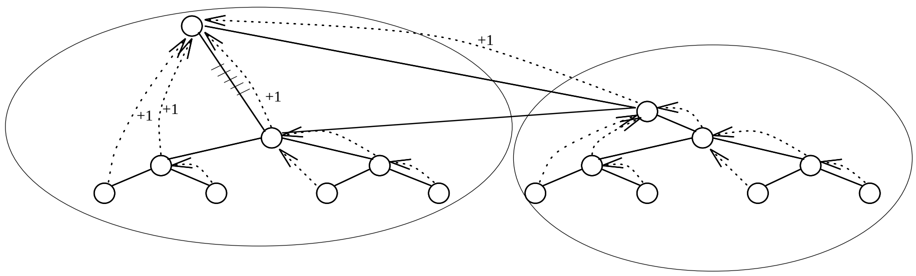

Abbildung 7: Analyse der Gesamtlänge der Pfade, die durch Gparent definiert sind.

Auf jedem Level verdoppelt sich die Gesamtlänge der Pfade und es kommt ein ein Pfad der Länge k hinzu, da für alle $v \in S_{root(T)}$ der Abstand von $Gparent(v)$ zu v um 1 gewachsen ist. Somit gilt die folgende Rekursion:

$$BO(2) = 1 \tag{3}$$
$$BO(2^k) = 2 \cdot BO(2^{k-1}) + k \tag{4}$$

und damit:

$$\begin{aligned}BO(2^k) &= 2^1 \cdot BO(2^{k-1}) + k \\ &= 2^1 \cdot (2 \cdot BO(2^{k-2}) + k - 1) + k \\ &= 2^2 \cdot (2 \cdot BO(2^{k-2}) + 2^1(k-1) + 2^0 k\end{aligned}$$

$$
\begin{aligned}
&= 2^{k-1} \cdot BO(2^1) + \sum_{j=0}^{k-2} 2^j (k-j) \\
&= n/2 + k \sum_{j=0}^{k-2} 2^j - \sum_{j=0}^{k-2} j 2^j \\
&= n/2 + k(2^{k-1} - 1) - (k-3)2^{k-1} - 2 \\
&= n/2 + k2^{k-1} - k - k2^{k-1} + 3n/2 - 2 \\
&= 2n - \log n - 2\Box
\end{aligned}
$$

Satz 5 *Die Initialisierung ist mit $n-1$ wesentlichen Vergleichen und $O(n)$ anderen Operationen effizient durchführbar.*

Beweis: (Dutton (1993)[5]) Offensichtlich wird für alle $j \in \{1, \dots n-1\}$ die Funktion $Merge(Gparent(j), j)$ aufgerufen. Sie benötigt jeweils einen Schlüsselvergleich.
Die Anzahl der Bitoperationen eines $Gparent(j)$-Aufrufes ist gleich der Entfernung von $Gparent(j)$ und j im Baum ($dist(Gparent(j), j)$),
da die Berechnung von $\lfloor j/2 \rfloor$ (j div 2) und $odd(j)$ (j mod $2 \neq 0$) als eine Maschinenoperation angesehen werden kann. Die Zahl dieser Operationen ist wiederum nach Satz 4 durch $O(n)$ beschränkt. □
Dennoch soll auf eine einfache von Dutton nicht betrachtete rekursive Implementation von $WeakHeapify$ hingewiesen werden, die die $(Gparent(j), j)$–Paare in anderer, aber zulässiger Reihenfolge besucht (Aufruf: $WeakHeapify^*(0)$).

```
PROCEDURE WeakHeapify*(h)
  j = 2h + 1
  WHILE 2j < n DO j = 2j
  WHILE j > h DO
    IF 2j + 1 < n THEN WeakHeapify*(j)
    Merge(h,j)
    j = j DIV 2
  OD
END WeakHeapify*.
```

Hierbei wird für jedes Element $j \in S_h$ die Funktion bottom-up rekursiv aufgerufen, falls überhaupt ein rechtes Kind existiert. Die Rekursionstiefe ist durch die Anzahl der Wechsel in den rechten Teilbaum, also durch $\log(n)$ beschränkt. Auch hier wird $Merge(h = Gparent(j), j)$ nur dann aufgerufen, wenn sowohl $Merge(Gparent(rchild(j)), rchild(j))$ als auch
$Merge(Gparent(lchild(j)), lchild(j))$ terminiert sind, da $WeakHeapify^*(h)$ rekursiv $WeakHeapify^*(j)$ für alle $j \in S_h$ aufruft, *bevor* $Merge(h, j)$ ausgeführt wird. Der Satz 2 findet demnach auch hier seine Anwendung zum Beleg der

[5]Ebd. ist die Anzahl der übrigen Operationen nicht analysiert worden.

Korrektheit. Die Anzahl der Bitoperationen bleibt durch $O(n)$ beschränkt, da jede Baumkante maximal zweimal (mittels Bitverschiebung und Inkrement um 1) besucht wird.

3.2.4 Die Funktion MergeForest

Falls das Array von $a[0], \ldots, a[m]$ einen Weak-Heap repräsentiert und $a[0]$ als Wurzelelement entnommen wurde, so reorganisiert die folgende Prozedur[6] den Weak-Heap mit neuer Wurzel $a[m]$:

```
PROCEDURE MergeForest(m)
  x = 1
  WHILE (2x + Reverse[x] < m) DO
    x = 2x + Reverse[x]
  OD
  WHILE x > 0 DO
    Merge(m,x)
    x = x DIV 2
  OD
END MergeForest.
```

In der ersten WHILE–Schleife wird der Weg beschrieben, der sich von dem rechten Sohn der Wurzel (Index: 1) ausgehend ergibt, wenn man sich stets zum linken Kind wendet und repräsentiert somit S_0. Sei dieser Pfad im folgenden spezieller Pfad (SP) gennant, und sei x das letzte Element auf diesem Pfad. Der Arrayindex m soll zur neuen Wurzel werden und darf somit nicht Element von SP sein. Die Arraydarstellung eines Weak-Heaps bewirkt, daß der höchste Index m nicht mehr frei gewählt werden kann, also ein bestimmtes Blatt repräsentiert. Es ergeben sich drei Fälle (vergl. Abb.8):

a) $depth(x) = depth(m)$,

b) $x = \lfloor m/2 \rfloor$,

c) $(depth(x) \neq depth(m)) \wedge (x \neq \lfloor m/2 \rfloor)$.

Da die Anzahl von Schlüsselvergleichen mit der Anzahl von Merge-Aufrufen übereinstimmt und diese sich wiederum aus der Tiefe von x berechnet, erhält man unmittelbar folgenden

Satz 6 *MergeForest benötigt in den Fällen b) und c)* $\lceil log(m+1) \rceil - 1$ *Vergleiche, im Fall a)* $\lceil log(m+1) \rceil$ *Vergleiche.*

[6] Die Originalarbeiten beinhalten eine unötige Zuweisung `Reverse[m DIV 2] = 0` und eine unnötige Abfrage `IF (m >= 3)`.

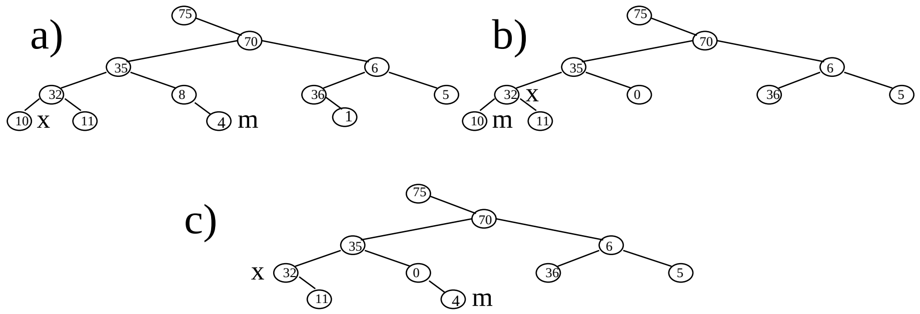

Abbildung 8: Die Lage von dem Knoten mit Index m zu dem mit Index x.

Beweis: (Dutton (1992)) In den Fällen b) und c) liegen m und x auf unterschiedlichen Leveln, d.h. $depth(x) = depth(m) - 1 = \lceil \log(m+1) \rceil - 1$, in Fall a) gilt hingegen $depth(x) = depth(m) = \lceil \log(m+1) \rceil$.□
Beweistechnisch ist es häufig nützlich, eine Variante $MergeForest^*(m)$ von $MergeForest(m)$ zu betrachten, bei der das Vergleichselement $a(m)$ vorher mittels $Swap(a[0], a[m])$ an die Stelle 0 getauscht wurde. Die Prozedur $MergeForest^*$ gestaltet sich dann wie folgt:

```
PROCEDURE MergeForest*(m)
  x = 1
  WHILE (2x + Reverse[x] < m) DO
    x = 2x + Reverse[x];
  OD
  WHILE x > 0 DO
    Merge(0,x)                          {* vorher: Merge(m,x) *}
    x = x DIV 2
  OD
END MergeForest*.
```

3.2.5 Die Funktion WeakHeapSort

Der Sortiervorgang ergibt sich nun aus einem einmaligen $WeakHeapify$ und aufeinanderfolgenden Aufrufen von $MergeForest$.

```
PROCEDURE WeakHeapSort
  WeakHeapify
  a[n] = a[0]
  FOR i = n-1 DOWNTO 2 DO MergeForest(i)
END WeakHeapSort.
```

Die Korrektheit dieses Verfahrens ergibt sich aus der Invarianz, daß nach dem Aufruf von $MergeForest(i)$ immer ein Weak-Heap der Größe i in dem Array

$a[1], \ldots, a[i]$ eingebettet ist. Allerdings repräsentiert $a[i]$ und nicht $a[0]$ die Wurzel. Somit liegt das i. - größte Element auch an Position i und muß im Anschluß nicht mehr als Wurzel entnommen werden.
Die Analyse der best- und worst-case Anzahl von Operationen gründet sich auf einige Vorbetrachtungen:

Hilfssatz 3

$$\sum_{i=1}^{n} \lceil \log i \rceil = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1. \tag{5}$$

Beweis: Als Idee dient die geschickte Aufteilung der Summe:

												n		
i	1	2	3	4	5	6	7	8	9	10	11	12	13	.....
⌈log i⌉	0	1	2	2	3	3	3	3	4	4	4	4	4	.....

2^0*1 2^1*2 2^2*3 $(n-2^3)*4$

Abbildung 9: Geeignetes Zusammenfassen von Summanden.

$$\begin{aligned}
\sum_{i=1}^{n} \lceil \log i \rceil &= \sum_{i=1}^{\lceil \log n \rceil - 1} i2^{i-1} + \lceil \log n \rceil (n - 2^{\lceil \log n \rceil - 1}) \\
&= \frac{1}{2}\Big(\sum_{i=1}^{\lceil \log n - 1 \rceil} i2^{i}\Big) + n\lceil \log n \rceil - \frac{1}{2}\lceil \log n \rceil 2^{\lceil \log n \rceil} \\
&\overset{HS2}{=} \frac{1}{2}((\lceil \log n \rceil - 2)2^{\lceil \log n \rceil} + 2) + n\lceil \log n \rceil - \frac{1}{2}\lceil \log n \rceil 2^{\lceil \log n \rceil} \\
&= \frac{1}{2}\lceil \log n \rceil 2^{\lceil \log n \rceil} - 2^{\lceil \log n \rceil} + 1 + n\lceil \log n \rceil - \frac{1}{2}\lceil \log n \rceil 2^{\lceil \log n \rceil} \\
&= n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1. \square
\end{aligned}$$

Folgerung 1 *Sei $k = \lceil \log n \rceil$, dann ist*

$$\sum_{i=2}^{n-1} (\lceil \log(i+1) \rceil - 1) = nk - 2^k - n + 2. \tag{6}$$

Beweis:

$$\sum_{i=2}^{n-1} (\lceil \log(i+1) \rceil - 1) = \sum_{i=3}^{n} (\lceil \log(i) \rceil - 1)$$

$$
\begin{aligned}
&= \sum_{i=3}^{n} \lceil \log(i) \rceil - (n-2) \\
&= \sum_{i=1}^{n} \lceil \log(i) \rceil - 1 - n + 2 \\
&= nk - 2^k - n + 2. \square
\end{aligned}
$$

Hilfssatz 4 *Das Maximum von* $f(x) = x - 2^x + 1, x \in [0,1]$ *liegt bei* $x_0 = -\frac{\ln \ln 2}{\ln 2}$ *und* $f(x_0) \approx 0.086013$. *Das Minimum der Funktion* $g(x) = f(x) - 1, x \in [0,1]$ *liegt bei* $x_1 = 0$ *mit* $g(x_1) = -1$.

Beweis:

$$
\begin{aligned}
f(x) &= x - e^{x \ln 2} + 1. \\
f'(x) &= 1 - e^{x \ln 2} \ln 2 \\
&= 1 - 2^x \ln 2. \\
f''(x) &= -2^x (\ln 2)^2.
\end{aligned}
$$

Nach dem notwendigen Kriterium für ein Extremum $f'(x_0) = 0$ ergibt sich:

$$
\begin{aligned}
1 - 2^{x_0} \ln 2 &= 0 \\
2^{x_0} &= \frac{1}{\ln 2} \\
x_0 \ln 2 &= \ln 1 - \ln \ln 2 \\
x_0 &= -\frac{\ln \ln 2}{\ln 2}.
\end{aligned}
$$

Für $x \in [0,1]$ ist $f''(x)$ kleiner als Null. Es gilt f(0) = f(1) = 0. Somit ist x_0 Maximalstelle und x_1 Minimalstelle von f (als auch g) mit oben angegebenen Funktionswerten.

Satz 7 *Sei* $k = \lceil \log n \rceil$. *Die minimale Anzahl von Schlüsselvergleichen von WEAK-HEAPSORT für alle* $n \geq 1$ *ist* $nk - 2^k + 1 \geq n \log n - n$ *und die maximale Anzahl von Schlüsselvergleichen von WEAK-HEAPSORT für alle* $n \geq 1$ *ist* $nk - 2^k + n - k \leq (n-1) \log n + 0.086013n$. *Der Unterschied zwischen dem best- und dem worst-case ist höchstens* $n - k - 1$.

Beweis: (Dutton (1992)) Da kein Vergleich für $n = 1$ notwendig ist, gelten die angegebenen Formeln. Ist $n \geq 2$, dann ergibt sich die minimale Anzahl von Vergleichen unter Ausschluß von $WeakHeapify$, wenn Fall a) nie eintritt, d.h. es gibt durch die Aufrufe $MergeForest(i)$ für $i = n-1, \ldots, 2$ mindestens

$$
\sum_{i=2}^{n-1} (\lceil \log(i+1) \rceil - 1) = nk - 2^k - n + 2 \qquad (7)
$$

Vergleiche. Zusammen mit den $n-1$ Vergleichen, die zum Aufbau des Heaps benötigt werden, sind dies mindestens $nk - 2^k + 1$ Vergleiche für den gesamten Sortieralgorithmus.
Für alle $n \in N$ existiert ein $x \in [0, 1[$, so daß $\lceil \log n \rceil = \log n + x$. Damit ist

$$\begin{aligned} nk - 2^k + 1 &= n \log n + nx - n2^x + 1 \\ &= n \log n + n(x - 2^x) + 1. \end{aligned}$$

Die reelle Funktion $g(x) = x - 2^x$ ist für $x \in [0, 1[$ nach unten durch -1 beschränkt. Damit ist die Anzahl an Vergleichen größer als $n \log n - n$.
Eine obere Schranke für die Anzahl von Vergleichen ergibt sich, falls für jeden Knoten i, $i = n-1, \ldots, 2$, Fall a) eintrifft. Allerdings tritt der günstige Fall b) oder c) mindestens einmal pro Level auf, nämlich wenn $m = 2^l$, $l = 1, \ldots, k$. Damit ist die worst-case Anzahl an Vergleichen durch $(nk - 2^k + 1) + n - 1 - k = nk - 2^k + n - k$ beschränkt. Für alle $n \in N$ existiert wieder ein $x \in [0, 1[$, so daß:

$$\begin{aligned} nk - 2^k + n - k &= n \log n + nx - n2^x + n - \log n - x \\ &= (n-1) \log n + n(x - 2^x + 1) - x. \end{aligned}$$

Die reelle Funktion $f(x) = x - 2^x + 1$ ist nach oben durch 0.086013 beschränkt. Damit ist die Anzahl an Vergleichen geringer als $(n-1)\log(n) + 0.086013n$.$\square$

4 Verbesserungen von WEAK-HEAPSORT

Dem Ersten gebührt der Ruhm, auch wenn die Nachfolger es besser gemacht haben.

Arabisches Sprichwort

Der vorgestellte WEAK-HEAPSORT Algorithmus nach Dutton birgt einige Verbesserungsmöglichkeiten, die im folgenden aufgezeigt und analysiert werden.

4.1 Entschärfung des worst-case

Die Wegschaffung des Schlimmen wird schon das Gute bringen.

Johann Gottfried Seume

Die Idee ist es, in einem zumindest im worst-case auftretenden Fall von dem ursprünglichen Algorithmus abzuweichen.

```
PROCEDURE MergeForest(m)
  x = 1
```

```
  WHILE 2x + Reverse[x] < m DO x = 2x + Reverse[x] OD
  IF (((m-1 = x) AND (m DIV 2 <> x)) AND (depth(x) = depth(m))
  THEN
    WHILE x > 1 DO
      Merge(m-1, x DIV 2)
      x = x DIV 2
    OD
    swap(a[m],a[m-1])
  ELSE
    WHILE x > 0 DO
      Merge(m,x)
      x = x DIV 2
    OD
  FI
END MergeForest.
```

Die beiden letzten Bedingungen der IF–Abfrage garantieren, daß der Fall a) (vergl. Abb.8) vorliegt, in dem sich die ungünstige Vergleichszahl von $\lceil\log(m+1)\rceil$ ergibt. In diesem Fall wird wiederum die Situtation $m-1=x$ herausgefiltert, d.h. x und m liegen im Array direkt nebeneinander. Anstatt m wird nun $m-1$ als Wurzelelement betracht und der SP verkürzt sich entsprechend Fall b) um 1. Zum Ende des Verschmelzvorganges mittels $Merge$ wird das maximale Element von Index $m-1$ auf den Platz m getauscht. Die Korrektheit des Ansatzes ergibt sich aus den folgenden Überlegungen:
Die Weak-Heap Struktur mag zwar nach dem Verlassen der Prozedur $MergeForest$ an der Stelle $m-1$ verletzt sein, doch wird dieser Index unmittelbar beim darauf folgenden Aufruf zum neuen Parameter m'. In den ursprünglichen Fällen a), b) als auch c) ergeben sich keine Probleme, da an den Wert von m' keine Anforderungen gestellt sind. Genauso unproblematisch ist der Fall, wenn die Spezialbehandlung mehrfach nacheinander auftritt, da das größte zu findende Element sich nicht an der Stelle m' befinden kann, die Elemente des SPs dominieren alle anderen Werte im Weak-Heap.

Satz 8 *Sei* $k = \lceil\log n\rceil$. *Bei dem verbesserten WEAK-HEAPSORT Algorithmus liegt die maximale Anzahl von Schlüsselvergleichen für alle* $n \geq 1$ *unter* $nk - 2^k + n - 2k$. *Im Vergleich zu Satz 7 bedeutet dies also eine Verbesserung um einen Summanden der Größe* k.

Beweis: Eine obere Schranke für die Anzahl von Vergleichen ergab sich, falls für jeden Knoten i, $i \in \{n-1,\ldots,2\}$, der Fall a) eintraf, jedoch mit der Ausnahme, daß Fall b) mindestens einmal pro Level vorliegen mußte, nämlich wenn $m = 2^l$, $l \in \{1,\ldots,k\}$.
Es wird gezeigt daß in dem Fall $m = 2^l + 1$, $l \in \{1,\ldots,k\}$, ein Vergleich eingespart wird.

In diesem Fall befinden sich nur noch zwei Knoten auf dem untersten Level: Der Knoten m und der Knoten $m-1$. Der spezielle Pfad endet auf dem Knoten $m-1$, da sonst Fall b) vorliegen würde. Demnach greift die Fallunterscheidung und anstelle von m wird der Weak-Heap anfangend an der Stelle $m-1$ reorganisiert. Es ergeben sich somit $\lceil \log(m+1) \rceil - 1$ anstatt $\lceil \log(m+1) \rceil$ Vergleiche. Dem worst-case wird also auf jedem Level ein weiteres Mal die Möglichkeit 'entzogen', $\lceil \log(m+1) \rceil$ Vergleiche zu erhalten. □
Die Tabellen 1 und 2 zeigen die Verteilungen der Vergleiche in dem alten bzw. in dem verbesserten Algorithmus auf. Dabei wurden zu $n \in \{2, \ldots, 12\}$ alle $n!$ Permutationen erzeugt und die Vergleiche gezählt. Für $n \in \{2, 3, 4\}$ fällt die obere (in der verbesserten Version gleich der unteren) Anzahl der benötigten Vergleiche für alle Eingabepermutationen mit der unteren Schranke $\lceil \log n! \rceil$ für allgemeine Sortierverfahren zusammen. Für größere Werte von n wird auf diese untere Schranke mit dem Kürzel US in den Tabellen hingewiesen.

4.2 Best-case bei linearem Zusatzspeicher

Wollen wir nicht würfeln, damit der Beste auch eine Chance hat?

Thomas Finkenstaedt

Die Idee wird ausgebaut: Jedesmal soll nun in dem Fall a) mit der Hilfe eines linear großen Arrays $(a[n+1], \ldots, a[2n])$ von dem ursprünglichen Algorithmus abgewichen werden. Da die Indizes auf dem letzten Level aufgrund der Belegung der Reversebits stark variieren, wird ein Boole'sches Array $Exist$ zur Kontrolle der noch gültigen Stellen verwaltet. Ist $Exist[i]$ wahr, so ist der Schlüssel an der Stelle i noch nicht in den Bereich $n+1, \ldots, 2n$ getauscht, sprich für den aktuellen Weak-Heap noch gültig. Der Sprung von einer gültigen Stelle zur nächsten wird durch eine Schleifenvariable *call* organisiert. Die $MergeForest$ Prozedur gestaltet sich wie folgt:

```
PROCEDURE MergeForest(m)
  x = 1
  WHILE 2x + Reverse[x] < m DO x = 2x + Reverse[x] OD
  IF ((m DIV 2 <> x) AND (depth(x) = depth(m))
  THEN
    temp = x
    WHILE x > 1 DO
      Merge(temp, x DIV 2)
      x = x DIV 2
    OD
    swap(a[2n - call],a[temp])
    exist[temp] = 0
  ELSE
```

	2	3	4	5	6	7	8	9	10	11	12
1	1										
2											
3		6									
4											
5			12								
6			12								
7				US							
8				48							
9				72							
10					US						
11					168						
12					408						
13					144	US					
14						504					
15						2040					
16						2136	US				
17						360	1008				
18							8064				
19							18144	US			
20							11952				
21							1152	5328			
22								60090	US		
23								163536			
24								120912			
25								13008	30816		
26									418176	US	
27									1394304		
28									1371552		
29									381984	179424	US
30									31968	2872128	
31										11374944	
32										15492480	
33										8283456	952128
34										1608960	17518656
35										105408	85879680
36											166160256
37											144379008
38											55165632
39											8531520
40											414720

Tabelle 1: Verteilung von Vergleichen nach Dutton.

	2	3	4	5	6	7	8	9	10	11	12
1	1										
2											
3		6									
4											
5			24								
6											
7				US							
8				120							
9											
10					US						
11					720						
12											
13						US					
14						3600					
15						1440					
16							US				
17							14400				
18							21600				
19							4320	US			
20											
21								100800			
22								210240	US		
23								51840			
24											
25									864000		
26									2139840	US	
27									624960		
28											
29										7488000	
30										21317760	
31										9892800	
32										1218240	
33											
34											
35											
36											
37											
38											
39											
40											

Tabelle 2: Verteilung von Vergleichen beim verbesserten Algorithmus.

```
    WHILE x > 0 DO
      Merge(m,x)
      x = x DIV 2
    OD
    Swap(a[2n - call], a[m])
    exist[m] = 0
  FI
END MergeForest.
```

Die Bedingungen der IF–Abfrage garantieren hier wiederum, daß der Fall a) vorliegt. Nun wird $temp = x$ zur ausgezeichneten Wurzelstelle, und der SP verkürzt sich entsprechend. Da genügend Zusatzspeicherplatz $(a[n], \ldots, a[2n])$ vorhanden ist, kann in allen Fällen das maximale Element dorthin geschrieben werden und die Stelle durch eine Markierung als für den Algorithmus nicht mehr existierend gekennzeichnet werden. Die äußere Schleife

```
  i = n-1
  call = 0
  WHILE i > 1 DO
    WHILE exist[i] = 0 DO dec(i)
    inc(call)
    MergeForest(i)
  OD
```

organisiert dann die Sortierung.
Die amortisierten Kosten (vergl. Cormen (1990)) der verschachtelten Schleife sind kleiner als $2n \in O(n)$, da pro Aufruf von $MergeForest$ nur ein Wert von $Exist$ auf Null gesetzt werden kann.
Der Algorithmus erreicht immer exakt seine best-case Vergleichsanzahl von $n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + n - \lceil \log n \rceil$.
Natürlich ist dieser Sortierungsalgorithmus eher von theoretischer Bedeutung, da ein linearer Zusatzspeicher für allgemeine Schlüsselvergleichsverfahren nicht zulässig ist.
Auch andere HEAPSORT–Varianten werden mit Zusatzspeicher in dieser Größenordnung bedeutend schneller: Die Lücke, die sich durch die Entnahme eines Wurzelelementes ergibt, kann z.B. durch einen einzigen Vergleich der Kinder untereinander geschlossen werden.

5 Die best-case Analyse von WEAK-HEAPSORT

Am Anfang der Forschung steht das Staunen. Plötzlich fällt einem etwas auf.

Wolfgang Wickler

Dieser Abschnitt liefert den Beleg, warum bei aufsteigender Vorsortierung des zu sortierenden Arrays der Größe n immer der best-case an Schlüsselvergleichen gegeben ist.
Damit nach der Generierung mittels $WeakHeapify$ der günstige Fall b) (vergl. Abb. 8) vorliegen kann, muß $n-1 \in S_{root(T)} = S_0 = SP$ liegen, d.h. das Routing mittels aufeinanderfolgenden $2i + Reverse[i]$– Schritten muß von dem Index 1 ausgehend an der Stelle $n-1$ enden. Hieraus läßt sich die Belegung von $Reverse[i]$ für $i \in SP$ aus der Binärcodierung von $n-1$ bestimmen. Sei dazu für einen Knotenindex v mit $b = bin(v)$ die zugehörige Binärdarstellung und umgekehrt zu gegebener Binärdarstellung $b = b_k \dots b_0$ mit $(b)_2 = (b_k \dots b_0)_2$ der Index in der ursprünglicher Darstellung bezeichnet.

Satz 9 *Sei $f : R \to R$ stetig monoton steigend mit der Eigenschaft:*

$$(*) \qquad f(x) \in Z \Longrightarrow x \in Z.$$

Dann gilt: $\lfloor f(x) \rfloor = \lfloor f(\lfloor x \rfloor) \rfloor$

Beweis: (Graham (1989)). Wenn $x = \lfloor x \rfloor$, dann ist nichts zu beweisen. Aus der Monotonie folgt für $x > \lfloor x \rfloor$: $f(x) > f(\lfloor x \rfloor)$. Da auch die untere Gaußklammer eine monoton nicht fallende Funktion ist gilt: $\lfloor f(x) \rfloor \geq \lfloor f(\lfloor x \rfloor) \rfloor$. Annahme: $\lfloor f(x) \rfloor > \lfloor f(\lfloor x \rfloor) \rfloor$. Dann ist $\lfloor f(x) \rfloor \geq \lfloor f(\lfloor x \rfloor) \rfloor + 1 > f(\lfloor x \rfloor)$. Sicherlich gilt: $\lfloor f(x) \rfloor \leq f(x)$. Somit ist aufgrund der Stetigkeit von f der Zwischenwertsatz anwendbar, d.h. es existiert ein y mit $\lfloor x \rfloor < y \leq x$ und $f(y) = \lfloor f(x) \rfloor$. Die Bedingung (*) zieht nach sich, daß y ganzzahlig seien muß, was im Widerspruch zur Wahl von y steht. □

Folgerung 2 *Sei $m \in R$, dann gilt:*

$$\lfloor \frac{\lfloor x \rfloor - m}{2} \rfloor = \lfloor \frac{x - m}{2} \rfloor.$$

Hilfssatz 5 *Sei $n-1 = (b_k \dots b_0)_2$ mit $k = \lfloor \log(n-1) \rfloor$. Wenn $Reverse[\lfloor \frac{n-1}{2^i} \rfloor] = b_{i-1}$ für $i \in \{1, \dots, k\}$ ist, dann ist $n-1$ das letzte Element von SP.*

Beweis: Die Menge $P = \{\lfloor \frac{n-1}{2^i} \rfloor \mid i \in \{1, \dots, k\}\}$ beschreibt den Weg von dem Index $n-1$ zurück zum Index 1, da

$$\lfloor \frac{\lfloor \frac{n-1}{2^{i-1}} \rfloor}{2} \rfloor = \lfloor \frac{n-1}{2^i} \rfloor \tag{8}$$

nach Folgerung 2 gilt und mit $\lfloor a/2 \rfloor$ jeweils das Elternteil zu a bestimmt wird. Weiterhin gilt ($\div$ steht mod, r für $Reverse$):

$$\begin{array}{lclclclcl}
n-1 \div 2 & = & b_0 & \Rightarrow & n-1 & = & 2\lfloor \frac{n-1}{2^1} \rfloor + b_0 & = & 2\lfloor \frac{n-1}{2^1} \rfloor + r[2\lfloor \frac{n-1}{2^1} \rfloor] \\
\lfloor \frac{n-1}{2^1} \rfloor \div 2 & = & b_1 & \Rightarrow & \lfloor \frac{n-1}{2^1} \rfloor & = & 2\lfloor \frac{n-1}{2^2} \rfloor + b_1 & = & 2\lfloor \frac{n-1}{2^2} \rfloor + r[2\lfloor \frac{n-1}{2^2} \rfloor] \\
\dots & = & \dots & \Rightarrow & \dots & = & \dots & = & \dots \\
\lfloor \frac{n-1}{2^{k-1}} \rfloor \div 2 & = & b_{k-1} & \Rightarrow & \lfloor \frac{n-1}{2^{k-1}} \rfloor & = & 2\lfloor \frac{n-1}{2^k} \rfloor + b_{k-1} & = & 2 \cdot 1 + r[1].
\end{array}$$

Damit wird jedoch der Spezielle Pfad SP (rückwärts gelesen) beschrieben und $n-1$ ergibt sich als dessen letztes Element, es ist $P = SP$.□

Hilfssatz 6 *Seien Knoten i, j mit $i \neq j$ und gleicher Höhe $height(i) = height(j)$ im Baum T gegeben, dann gilt $Gparent(i) \neq Gparent(j)$.*

Beweis: Annahme, es gäbe $i, j \in T$ mit $i \neq j$ und $height(i) = height(j)$ und $Gparent(i) = Gparent(j)$. Dann ergeben sich folgenden Möglichkeiten:

i linkes und j rechtes Kind: Per Definition der Relation *Gparent* gilt somit aber $height(i) > height(j)$, Widerspruch.

i rechtes und j linkes Kind: Analog.

i rechtes und j rechtes Kind: Per Definition der Relation *Gparent* gilt dann aber $Parent(i) = Parent(j)$ und somit $i = j$, Widerspruch.

i linkes und j linkes Kind: Definiere $i' := Parent(i)$ und $j' := Parent(j)$. Falls $i' = j'$, dann ist auch $i = j$, Widerspruch. Sonst iteriere die Fallunterscheidung mit i' und j', Wenn sich für sie ein Widerspruch ergibt, so auch für i und j.

Spätestens an der Wurzel kann der letzte Fall nicht mehr eintreten, d.h. die Annahme ist widerlegt und der Satz bewiesen.□
Diese Aussage ließe sich auch daraus schlußfolgern, daß die Menge S_i aus den Knoten, die i als gemeinsamen *Gparent* besitzen, einen Pfad bilden.
Nun kann gezeigt werden, daß die Bedingung von Hilfsatz 5 bei einem vorsortierten Array nach Abschluß der Generierungsphase erfüllt sind.

Hilfssatz 7 *Sei $n-1 = (b_k \dots b_0)_2$ mit $k = \lfloor \log(n-1) \rfloor$, $P = \{\lfloor (n-1)/2^i \rfloor \mid i \in \{1, \dots, k\}\}$ der Weg von Index $n-1$ zu Index 1. Bei einer Initialkonfiguration $a[i] = i, i \in \{0, \dots, n-1\}$, gilt nach Beendigung von $WeakHeapify$:*

a) $Reverse[0] = 0$,

b) $Reverse[j] = 1$, $j \notin P$,

c) $Reverse[n-1] = 1$ *und* $Reverse[\lfloor \frac{n-1}{2^i} \rfloor] = b_{i-1}$, $i \in \{1, \dots, k\}$.

Beweis:

zu a) $Reverse[0]$ kann nur 0 sein, da $Merge(i, j)$ nur das Reversebit des zweiten Parameters setzt und $Merge(Gparent(0), 0)$ nicht aufgerufen wird.

zu b) Mit $j \in \{\lfloor\frac{n-1}{2}\rfloor + 1, \ldots, n-1\}$ werden die Indizes der Blätter des initialen Binärbaumes beschrieben. Aufgrund der Vorsortierung gilt für alle i, j mit $i < j : a[i] < a[j]$, d.h. insbesondere für alle j: $a[Gparent(j)] < a[j]$. Alle Blätter haben die gleiche Höhe 0 im Baum, so daß sich nach Hilfssatz 6 für paarweise verschiedene Blätter auch paarweise verschiedene Großeltern ergeben. Mit eventueller Ausnahme von $\lfloor\frac{n-1}{2}\rfloor$ (n ungerade) wechseln somit alle Werte von inneren Knoten in die Blätter und umgekehrt. Das Reversebit wird also für die Blätter gesetzt. Da

$$Gparent(j) = \begin{cases} Gparent(Parent(j)) & \text{falls } j \text{ linkes Kind,} \\ Parent(j) & \text{falls } j \text{ rechtes Kind} \end{cases}$$

gilt, sind die Großeltern aller Blätter, die linke Kinder sind, per Definition die Großeltern der Großeltern ihrer rechten Geschwister. Aufgrund der Vorsortierung gilt demnach außer für die Blätter wieder $a[Gparent(j)] < a[j]$. Da die Elemente der Höhe 1 sich jeweils aus den rechten Blattkindern durch Vertauschung ergeben, gilt auch für diese Elemente, daß die rechten Knoten ihre linken Geschwister dominieren. Damit läßt sich das obige Verfahren jedoch iterieren und alle Reversebits für Elemente $j \notin P$ werden gesetzt.

zu c) Für Elemente j aus dem durch $n-1$ beschriebenen Pfad P sind die durch j beschriebenen Teilbäume nicht immer vollständig und somit nicht immer linke und rechte Kinder vorhanden. Betrachte nun den Pfad P bottom–up. Das Reversebit des Blattes $n-1$ wird gesetzt, d.h. $Merge(Gparent(n-1), n-1)$ wird durchgeführt. Falls $n - 1 \div 2 = b_0 = 1$ ($\div$ steht wieder für mod), so führt der Weg vom rechten Kind zum Elternteil (Fall i)) und sonst vom linken Kind(Fall(ii)):

i) Keine Änderung der Argumentation zu Fall b), d.h. das Reversebit wird auch noch in der nächsten Stufe gesetzt.

ii) Nun dominiert $a[Gparent(n-1)] = n-1$ alle folgenden Elemente des Pfades P, insbesondere $a[\lfloor(n-1)/2\rfloor]$. Demnach kann $a[\lfloor(n-1)/2\rfloor]$ nicht mehr mit $a[Gparent(n-1)]$ getauscht werden, d.h. $Reverse[\lfloor(n-1)/2\rfloor]$ wird auf 0 gesetzt.

Für den weiteren Verlauf des Pfades P läßt sich wie folgt analog schließen: Führt der Weg von einem rechten Kind $\lfloor(n-1)/2^{i-1}\rfloor$ zum Elternteil $\lfloor(n-1)/2^i\rfloor$, d.h. $\lfloor(n-1)/2^{i-1}\rfloor \div 2 = b_{i-1} = 1$, so wird das Reversebit an der Stelle $\lfloor(n-1)/2^i\rfloor$ gesetzt. Führt hingegen der Weg vom linken Kind $\lfloor(n-1)/2^{i-1}\rfloor$ zum Elternteil $\lfloor(n-1)/2^i\rfloor$, d.h. $\lfloor(n-1)/2^{i-1}\rfloor \div 2 = b_{i-1} = 0$, so kann es an der Stelle $\lfloor(n-1)/2^i\rfloor$ nicht gesetzt werden, da $a[Gparent(\lfloor(n-1)/2^{i-1}\rfloor)]$ alle folgenden Elemente des Pfades P, insbesondere $a[\lfloor(n-1)/2^{i-1}\rfloor]$ dominiert. In allen Fällen gilt $Reverse[\lfloor(n-1)/2^i\rfloor] = b_{i-1}$. $\square$

Hilfssatz 8 *Sei* $n-1=(b_k\dots b_0)_2$ *mit* $k=\lfloor\log(n-1)\rfloor$. *Es gilt:*

$$Gparent(n-1)=\lfloor\frac{n-1}{2^{i^*}}\rfloor \quad , \ mit \quad i^*=max\{i \mid b_{i-1}\neq 0\}$$

und

$$Gparent(\lfloor\frac{n-1}{2^{i^*}}\rfloor)=\lfloor\frac{n-1}{2^{j^*}}\rfloor \quad , \ mit \quad j^*=max\{j>i^* \mid b_{j-1}\neq 0\}.$$

Beweis: Nach Definition von *Gparent* gilt:

$$Gparent(\lfloor\frac{n-1}{2^i}\rfloor)=\begin{cases} Gparent(\lfloor\frac{n-1}{2^{i+1}}\rfloor) & \text{falls } b_i=\lfloor\frac{n-1}{2^i}\rfloor\div 2=1\\ \lfloor\frac{n-1}{2^{i+1}}\rfloor & \text{falls } b_i=\lfloor\frac{n-1}{2^i}\rfloor\div 2=0\end{cases}$$

und somit folgt die Behauptung induktiv. □

Hilfssatz 9 *Sei* $n\in N$ *mit* $n-1=(b_{n,k_n}\dots b_{n,0})_2$ *und* $k_n=\lfloor\log(n-1)\rfloor$, *sowie* $i^*=max\{i \mid b_{n,i-1}\oplus b_{n-1,i-1}=1\}$ *($\oplus$ steht für EXOR). Dann gilt:*

1. *Für alle* $j\leq i^*$: $b_{n,j-1}\oplus b_{n-1,j-1}=1$.
2. *Für alle* $j>i^*$: $b_{n,j-1}\oplus b_{n-1,j-1}=0$.

Beweis: Eine Betrachtung der Binäraddition von $n-2=(b_{n-1,k_{n-1}}\dots b_{n-1,0})_2$ mit $(d_{k_n},\dots,d_0)_2=(0\dots0\ \ 1)_2$ liefert nach der Schulmethode für die Stelle i: $b_{n,i}=b_{n-1,i}\oplus d_i\oplus c_{i-1}$, wobei c_i der i-te Übertrag ist und $c_{-1}=0$. Somit wird c_i genau an der ersten Stelle i^* verschluckt, wo $b_{n-1,i}=0$ gilt. Bis dorthin ist $b_{n-1,j}\neq b_{n,j}=1$ und ab dort ist $b_{n-1,j}=b_{n,j}$. Demnach ist $i^*=max\{i \mid b_{n,i-1}=0\}=max\{i \mid b_{n,i-1}\oplus b_{n-1,i-1}=1\}$. □

Hilfssatz 10 *Sei* $n\in N$ *mit* $n-1=(b_{n,k_n}\dots b_{n,0})_2$ *und* $k_n=\lfloor\log(n-1)\rfloor$. *Sei desweiteren* $a[i]=i$ *für* $i\in\{0,\dots,n-1\}$ *eine Eingabe für WEAK-HEAPSORT. Nach dem Aufruf der Prozedur* $Heapify$ *findet sich das Element* $n-2$ *an der Stelle* $\lfloor\frac{n-1}{2^{i^*}}\rfloor$ *mit* $i^*=max\{i \mid b_{n,i-1}\oplus b_{n-1,i-1}=1\}$.

Beweis: Fallunterscheidung:

$n-1\neq 2^k$**:** Es ist $k=k_n=k_{n-1}$ und vereinfachenderweise $n-1=(b_k\dots b_0)_2=(b_{n,k_n}\dots b_{n,0})_2$ und $n-2=(c_k\dots c_0)_2=(b_{n-1,k_{n-1}}\dots b_{n-1,0})_2$.

Im Fall $b_0=1$ ist $n-1$ rechtes und $n-2$ linkes Kind, also $Parent(n-2)=Parent(n-1)=Gparent(n-1)$ und somit

$$\begin{aligned} Gparent(n-2) &= Gparent(Parent(n-2))\\ &= Gparent(Parent(n-1))\\ &= Gparent(Gparent(n-1)). \end{aligned}$$

Somit gelangt das Element $n-2$ über die Position $Gparent(n-2)$ letztendlich an die Position $Gparent(n-1) = \lfloor \frac{n-1}{2^1} \rfloor$, da es dort mit dem Element $n-1$ verglichen wird. Andererseits ist $b_0 \oplus c_0 = 1$ und für alle $i \in \{1, \ldots, n-1\}$ gilt: $b_i = c_i$. Demnach ist $i^* = max\{i \mid b_{i-1} \oplus c_{i-1} = 1\} = 1$.

Im Fall $b_0 = 0$ ist $n-1$ linkes und $n-2$ rechtes Kind. Genauer: Es ist $n-1$ so lange linkes Kind, wie $n-2$ rechtes Kind ist, d.h. es gilt so lange $Parent(\lfloor \frac{n-1}{2^{i-1}} \rfloor) \neq Parent(\lfloor \frac{n-2}{2^{i-1}} \rfloor)$ wie $c_{i-1} = 1$ und $b_{i-1} = 0$ sind, also bis zu der Stelle $\lfloor \frac{n-1}{2^{i^*-1}} \rfloor$ mit $i^* = max\{i \mid b_{i-1} \oplus c_{i-1} = 1\} = max\{i \mid b_{i-1} \neq 0\}$. Dort ist $Parent(\lfloor \frac{n-2}{2^{i^*-1}} \rfloor) = Parent(\lfloor \frac{n-1}{2^{i^*-1}} \rfloor)$ also $\lfloor \frac{n-2}{2^{i^*}} \rfloor = \lfloor \frac{n-1}{2^{i^*}} \rfloor$.

Nach Hilfssatz 8 ist $Gparent(n-1) = \lfloor \frac{n-1}{2^{i^*}} \rfloor$. Sei $j^* = max\{j > i^* \mid b_{j-1} \neq 0\}$. Es gilt:

$$\begin{aligned} Gparent(Gparent(n-1)) &= Gparent(\lfloor \frac{n-1}{2^{i^*}} \rfloor) \\ &= \lfloor \frac{n-1}{2^{j^*}} \rfloor \\ &= Gparent(\lfloor \frac{n-2}{2^{i^*-1}} \rfloor). \end{aligned}$$

Somit gelangt das Element $n-2$ über die Positionen $\lfloor \frac{n-2}{2^j} \rfloor, j \in \{1, \ldots i^*-1\}$ und $\lfloor \frac{n-1}{2^{j^*}} \rfloor$ schließlich an die Position $\lfloor \frac{n-1}{2^{i^*}} \rfloor$, da es mit dem dann dortstehenden Element $n-1$ verglichen wird und folglicherweise die Plätze tauschen muß.

$n-1 = 2^k$: Es ist $k = k_n = k_{n-1} + 1$ und $n-1 = (b_k \ldots b_0)_2 = (b_{n,k_n} \ldots b_{n,0})_2$, $n-2 = (c_{k-1} \ldots c_0)_2 = (b_{n-1,k_{n-1}} \ldots b_{n-1,0})_2$. Weiterhin ist $n-1 = (1\ 0\ \ldots\ 0)_2$, und somit gilt nach Hilfssatz 8 $Gparent(n-1) = 0$. Da für das Element $n-2 = (c_{k-1} \ldots c_0)_2 = (1 \ldots 1)_2$ immer $Parent(\lfloor \frac{n-2}{2^{i-1}} \rfloor) = Gparent(\lfloor \frac{n-2}{2^{i-1}} \rfloor)$ gilt, landet $n-1$ erst vor dem Vergleich $Gparent(1) = 0 = (0)_2$ mit $1 = (1)_2$ auf dem durch $n-1$ beschriebenen Pfad $SP = \{\lfloor \frac{n-1}{2^i} \rfloor \mid i \in 0, \ldots, k\}$. In Übereinstimmung mit $i^* = k-1$ verbleibt es an der Position $1 = \lfloor \frac{n-1}{2^{k-1}} \rfloor$. □

Hilfssatz 11 *Sei $n \in N$ mit $n-1 = (b_{n,k_n} \ldots b_{n,0})_2$ und $k_n = \lfloor \log(n-1) \rfloor$. Sei desweiteren $a[i] = i$ für $i \in \{0, \ldots, n-1\}$ eine Eingabe für WEAK-HEAPSORT. Sei weiterhin $i^* = max\{i \mid b_{n,i-1} \oplus b_{n-1,i-1} = 1\}$. Nach dem Aufruf der Prozedur Heapify gilt folgende Monotonieeigenschaft: $a[\lfloor \frac{n-1}{2^j} \rfloor] > a[\lfloor \frac{n-1}{2^i} \rfloor]$ für $j < i \leq i^*$.*

Beweis: Die Menge $SP = \{\lfloor \frac{n-1}{2^i} \rfloor \mid i \in \{0, \ldots, k\}\}$ beschreibt den Weg von dem Index $n-1$ zurück zum Index 1. Nach Hilfssatz 5 ist $n-1$ das letzte Element von SP, also gilt $P = SP$.
Sei $n-1$ rechtes Kind. Dann wird es mit dem Element $\lfloor \frac{n-1}{2^1} \rfloor + 1$ an die Stelle $\lfloor \frac{n-1}{2^1} \rfloor$ getauscht. Es gilt: $1 = i^* = max\{i \mid b_{n,i-1} \oplus b_{n-1,i-1} = 1\}$, d.h nach

Hilfssatz 10 gelangt das Element $n-2$ in der nächsten Stufe an die Stelle $\lfloor \frac{n-1}{2^1} \rfloor$. Nun ist der Schlüssel $n-2$ an Stelle $\lfloor \frac{n-1}{2^1} \rfloor$ größer als $\lfloor \frac{n-1}{2^1} \rfloor + 1$, welches ja an Stelle $n-1$ steht. Daher gilt nach dem Aufruf der Prozedur $Heapify$ für $j < i = i^*$ die Eigenschaft $a[\lfloor \frac{n-1}{2^j} \rfloor] > a[\lfloor \frac{n-1}{2^i} \rfloor]$.
Ist $n-1$ linkes Kind, so gelangt nach Hilfssatz 10 das zweitgrößte Element $n-2$ nach der Generierungsphase an die Stelle $\lfloor \frac{n-1}{2^{i^*}} \rfloor = Gparent(n-1)$ mit $i^* = max\{i \mid b_{n,i-1} \oplus b_{n-1,i-1} = 1\}$. Sei $i < i^*$ gegeben, d.h. $\lfloor \frac{n-1}{2^i} \rfloor \in SP$. Dann muß jeweils das größte Element des rechten Teilbaumes an die Stelle $\lfloor \frac{n-1}{2^i} \rfloor$ befördert werden, da sonst die Weak-Heap Eigenschaft $(W1)$ nicht zutreffen kann. Deshalb gilt: $a[\lfloor \frac{n-1}{2^i} \rfloor] = max\{k \mid k \in rT(\lfloor \frac{n-1}{2^i} \rfloor)\} = (\lfloor \frac{n-1}{2^i} \rfloor + 1)2^i$. Man beachte, daß die durch $\lfloor \frac{n-1}{2^i} \rfloor$ beschriebenen Teilbäume vollständig sind und wegen der Vorsortierung das größte Element in $rT(\lfloor \frac{n-1}{2^i} \rfloor)$ in der Höhe 0 äußerst rechts zu finden ist. Damit gilt jedoch für $j < i < i^*$ die Eigenschaft $a[\lfloor \frac{n-1}{2^j} \rfloor] > a[\lfloor \frac{n-1}{2^i} \rfloor]$. Weil $a[\lfloor \frac{n-1}{2^{i^*}} \rfloor] = n-2$ ist, gilt diese Eigenschaft auch für $j < i = i^*$. □

Folgerung 3 *Sei $n \in N$ mit $n-1 = (b_{n,k_n} \ldots b_{n,0})_2$ mit $k_n = \lfloor \log(n-1) \rfloor$. Sei desweiteren $a[i] = i$ für $i \in \{0, \ldots, n-1\}$ eine Eingabe für WEAK-HEAPSORT. Nach dem Aufruf der Prozedur $Heapify$ ergibt sich im ersten $MergeForest^*$-Schritt die folgenden Kette von Transpositionen:*

$$\begin{aligned}&\tau^{b_{n,0} \oplus b_{n-1,0}}_{Gparent(\lfloor \frac{n-1}{2^1} \rfloor)=0, \lfloor \frac{n-1}{2^1} \rfloor}\\ &\circ \ldots \circ\\ &\tau^{b_{n,k_n-1} \oplus b_{n-1,k_n-1}}_{Gparent(\lfloor \frac{n-1}{2^{k_n}} \rfloor)=0, \lfloor \frac{n-1}{2^{k_n}} \rfloor = 1},\end{aligned}$$

wobei $\tau^a_{i,j}$ die a-mal angewendete Transposition von den Stellen i und j für $i, j \in \{0, \ldots, n-1\}$ bezeichnet und die Transpositionen auf eine links stehende Eingabe angewendet werden.

Beweis: Nach Hilfssatz 7 in Verbindung mit Hilfssatz 5 tritt bei Vorsortierung im ersten $MergeForest^*$-Schritt Fall b) ein.
Die Monotonie $a[\lfloor \frac{n-1}{2^j} \rfloor] > a[\lfloor \frac{n-1}{2^i} \rfloor]$ für $j < i < i^*$ besagt, daß alle Vertauschungen auf den rückwärts betrachteten Pfad SP von $a[\lfloor \frac{n-1}{2^j} \rfloor]$ und a[0] durchgeführt werden, da an der Wurzel immer ein kleineres Element steht als das nächst folgende auf SP. Nach Hilfssatz 9 gilt für diese j aber auch $b_{n,j-1} \oplus b_{n-1,j-1} = 1$, d.h. die zugehörige Transposition erscheint in der Kette.
An der Stelle $\lfloor \frac{n-1}{2^{i^*}} \rfloor$ liegt das maximale Element $n-2$. Es wird an die Wurzel getauscht. Per Definition von i^* gilt aber auch $b_{n,i^*-1} \oplus b_{n-1,i^*-1} = 1$.
Alle nun folgenden Elemente von SP können nicht mehr an die Wurzel gelangen. Dies steht in Übereinstimmung mit Hilfssatz 9, der sagt, daß für alle $j > i^*$: $b_{n,j-1} \oplus b_{n-1,j-1} = 0$ gilt.
Somit sind die sich durch den Fall b) ergebenden Vertauschungen korrekt durch die angegebenen Transpositionen beschrieben. □
Zusammenfassend gilt:

Satz 10 *Sei $n \in N$ mit $n-1 = (b_{n,k} \dots b_{n,0})_2$ mit $k_n = \lfloor \log(n-1) \rfloor$. Desweiteren definiere die folgenden Permutationen:*

$$
\begin{aligned}
&Heapi(n) := \tau^1_{Gparent(n-1),n-1} \circ \tau^1_{Gparent(n-2),n-2} \circ \dots \circ \\
&\tau^{b_{n,0}}_{Gparent(\lfloor \frac{n-1}{2^1} \rfloor),\lfloor \frac{n-1}{2^1} \rfloor} \circ \tau^1_{Gparent(\lfloor \frac{n-1}{2^1} \rfloor -1),\lfloor \frac{n-1}{2^1} \rfloor -1} \circ \dots \circ \\
&\tau^{b_{n,1}}_{Gparent(\lfloor \frac{n-1}{2^2} \rfloor),\lfloor \frac{n-1}{2^2} \rfloor} \circ \tau^1_{Gparent(\lfloor \frac{n-1}{2^2} \rfloor -1),\lfloor \frac{n-1}{2^2} \rfloor -1} \circ \dots \circ \\
&\vdots \\
&\tau^{b_{n,k_n-2}}_{Gparent(\lfloor \frac{n-1}{2^{k_n-1}} \rfloor),\lfloor \frac{n-1}{2^{k_n-1}} \rfloor} \circ \\
&\tau^{b_{n,k_n-1}}_{0,1}
\end{aligned}
$$

und $Swap(0, n-1) := \tau_{0,n-1}$, letztendlich

$$
\begin{aligned}
&CaseB(n) := \tau^{b_{n,0} \oplus b_{n-1,0}}_{Gparent(\lfloor \frac{n-1}{2^1} \rfloor)=0,\lfloor \frac{n-1}{2^1} \rfloor} \circ \\
&\vdots \\
&\tau^{b_{n,k_n-1} \oplus b_{n-1,k_n-1}}_{0,1},
\end{aligned}
$$

wobei $\tau^a_{i,j}$ die a-mal angewendete Transposition von den Stellen i und j für $i, j \in \{0, \dots, n-1\}$ bezeichnet und die Transpositionen auf eine links stehende Eingabe angewendet werden.
Sei desweiteren $a[i] = i$ für $i \in \{0, \dots, n-1\}$ eine Eingabe für WEAK-HEAP-SORT.
Dann werden die in der Aufbauphase durchgeführten Vertauschungen korrekt durch $Heapi(n)$ und die in dem ersten Auswahlschritt durchgeführten Vertauschungen korrekt durch die Hintereinanderausführung von $Swap(0, n-1)$ und $CaseB(n)$ beschrieben.

Beweis: Hilfssatz 7 belegt die richtige Beschreibung der Aufbauphase $Heapify$ bei gegebener Vorsortierung durch $Heapi(n)$.
Die Wurzelentnahme wird durch $Swap(0, n-1)$ geregelt, d.h. das maximale an der Wurzel stehende Element wird an die Stelle $n-1$ getauscht und im weiteren Verlauf nicht mehr berücksichtigt.
Nach dem Hilfsatz 3 wird der sich nach Hilfssatz 5 auftretende Fall b) durch $CaseB(n)$ korrekt beschrieben. □

Hilfssatz 12 *Sei ein Weak-Heap der Größe n gegeben und P der Pfad von der Wurzel zu einen Blatt.*
Es gilt für alle $x \in P$ und alle $y \notin P$:

1. *$Gparent(x) \neq y$.*

2. *Wenn $y < x$ ist, dann ist auch $Gparent(x) \neq Gparent(y)$.*

Beweis:

zu 1: Da

$$Gparent(x) = \begin{cases} Gparent(Parent(x)) & \text{falls } x \text{ linkes Kind} \\ Parent(x) & \text{falls } x \text{ rechtes Kind} \end{cases}$$

ist induktiv mit $x \in P$ auch $Gparent(x) \in P$.

zu 2: Wenn $Gparent(y) \notin P$, folgt $Gparent(x) \neq Gparent(y)$ unmittelbar nach 1. Sei also $Gparent(y) \in P$ (*). Es ergeben sich zwei Fälle:

y **liegt in einem rechten Teilbaum** T **von** P , d.h. es existiert ein $r \in P$ mit $root(T) = rchild(r)$. Für alle $w \in T$ gilt $w \notin P$. Das hat zweierlei Konsequenzen: Einerseits ist wegen (*) $y \in S_r$ und andererseits gilt für alle $v \in S_r : v \notin P$. Damit gilt für alle $x \in P : Gparent(x) \neq Gparent(y)$.

y **liegt in einem linken Teilbaum** T **von** P , d.h. es existiert ein $r \in P$ mit $root(T) = lchild(r)$. Für alle $w \in T$ gilt $w \notin P$. Damit ist $Gparent(y) = Gparent(r)$. Der Index von r in $S_{Gparent(r)}$ ist aber maximal mit der Eigenschaft, in P zu liegen. Damit gilt für alle $x \in P$ mit $x > y : Gparent(x) \neq Gparent(y)$.□

Folgerung 4 *Für alle* $p \notin SP = \{\lfloor \frac{n-1}{2^i} \rfloor \mid i \in \{0, \ldots, \lfloor \log(n-1) \rfloor\}$ *gilt:*

1. $p \neq n - 1 \neq Gparent(p)$,

2. $p \neq Gparent(n-1) \neq Gparent(p)$,

3. $p \neq \lfloor \frac{n-1}{2^i} \rfloor$ *und wenn* $\lfloor \frac{n-1}{2^i} \rfloor > p$ *ist, dann gilt* $\lfloor \frac{n-1}{2^i} \rfloor \neq Gparent(p)$,

4. $p \neq Gparent(\lfloor \frac{n-1}{2^i} \rfloor)$ *und wenn* $\lfloor \frac{n-1}{2^i} \rfloor > p$ *ist, dann gilt* $Gparent(\lfloor \frac{n-1}{2^i} \rfloor) \neq Gparent(p)$.

Beweis: Da $n-1$, $Gparent(n-1)$, $\lfloor \frac{n-1}{2^i} \rfloor$ und $Gparent(\lfloor \frac{n-1}{2^i} \rfloor) \in SP$ sind, gilt: $p \notin \{n-1, Gparent(n-1), \lfloor \frac{n-1}{2^i} \rfloor, Gparent(\lfloor \frac{n-1}{2^i} \rfloor\}$.

1. Da $n - 1 > p > Gparent(p)$ ist, gilt: $n - 1 \neq Gparent(p)$.

2. Nach Hilfssatz 12 und der Ungleichung aus 1. gilt: $Gparent(n - 1) \neq Gparent(p)$.

3. Es ist $\lfloor \frac{n-1}{2^i} \rfloor > p > Gparent(p)$.

4. Nach Hilfssatz 12 und der Ungleichung aus 3. gilt: $Gparent(\lfloor \frac{n-1}{2^i} \rfloor) \neq Gparent(p)$.□

Die Transposition von den Stellen i und j ($\tau_{i,j}$) wird im folgenden durch den Zykel $(i \;\; j)$ beschrieben werden. Die Beschreibung einer Permutation durch paarweise disjunkte Zykel ist eindeutig.

Definition 4 *Die Transpositionen $(i \;\; j)$ und $(k \;\; l)$ kollidieren, wenn $\{i,j\} \cap \{k,l\} \neq \emptyset$.*

Hilfssatz 13

$$\begin{aligned} (i \;\; j) \circ (j \;\; k) &= (i \;\; k) \circ (i \;\; j), \\ (j \;\; k) \circ (i \;\; j) &= (i \;\; j) \circ (i \;\; k). \end{aligned}$$

Beweis: Die Zusammenfassung in 3er–Zykel offenbart:

$$\begin{aligned} (i \;\; j) \circ (j \;\; k) &= (i \;\; j \;\; k) &= (i \;\; k) \circ (i \;\; j), \\ (j \;\; k) \circ (i \;\; j) &= (i \;\; k \;\; j) &= (i \;\; j) \circ (i \;\; k). \square \end{aligned}$$

Hilfssatz 14

$$Heapi(n) \circ (0 \;\; n-1) = (Gparent(n-1) \;\; n-1) \circ (Heapi(n)). \qquad (9)$$

Beweis: Idee: Die Transposition $(Gparent(n-1) \;\; n-1)$ soll sukzessiv durch die Kette der Transpositionen in $Heapi(n)$ *geschleift* werden. Dabei soll *schleifen* bedeuten, daß die Transpositionen in $Heapi(n)$ unverändert bleiben sollen und $(Gparent(n-1) \;\; n-1)$ gemäß Hilfssatz 13 propagiert werden soll.
Bezeichne mit τ_i^*, $i \in \{n-1, \ldots, 1\}$, die Transposition, die sich aus $\tau_n^* = (Gparent(n-1) \;\; n-1)$ nach der Verbindung von $(Gparent(i) \;\; i)$ und τ_{i+1}^* ergibt. Es ist zu zeigen, daß $\tau_1^* = (0 \;\; n-1)$.
Betrachte als erstes die Elemente der Höhe 0. Die Transposition
$\tau_n^* = (Gparent(n-1) \;\; n-1)$ ist gleich der ersten Transposition in $Heapi(n)$. Setze somit auch $\tau_{n-1}^* = (Gparent(n-1) \;\; n-1)$.
Für die Elemente $k \in \{n-2, \ldots, \lfloor \frac{n-1}{2} \rfloor - 1\}$ ergibt sich nach Folgerung 4 keine Kollision zwischen $\tau_{k+1}^* = \tau_{n+1}^*$ und $(Gparent(k) \;\; k)$.
Betrachte nun die Elemente der Höhe 1. Für das Element $\lfloor \frac{n-1}{2} \rfloor$ ergibt sich nach Hilfssatz 8 eine Kollision, wenn $b_0 \neq 0$ ist.
Dann setze $\tau^*_{\lfloor \frac{n-1}{2} \rfloor} = (Gparent(\lfloor \frac{n-1}{2} \rfloor) \;\; n-1)$, sonst setze $\tau^*_{\lfloor \frac{n-1}{2} \rfloor} = \tau^*_{n-1}$.

Für die Elemente $k \in \{\lfloor \frac{n-1}{2} \rfloor + 1, \dots \lfloor \frac{n-1}{4} \rfloor - 1\}$ ergibt sich nach Folgerung 4 wiederum keine Kollision zwischen τ^*_{k+1} und $(Gparent(k)\ \ k)$.
Nach Hilfssatz 8 verändert sich τ^* für die Höhen $i = 2, \dots, \lfloor \log(n-1) \rfloor$ nur bei den Elementen $\lfloor \frac{n-1}{2^i} \rfloor$, bei denen $b_{i-1} \neq 0$ gilt. Also folgt:

$$
\begin{aligned}
\tau^*_{\lfloor \frac{n-1}{2^{i_1}} \rfloor} &= (Gparent(\lfloor \frac{n-1}{2^{i_1}} \rfloor)\ \ n-1) \quad \text{für} \quad i_1 = max\{i \mid b_{i-1} \neq 0\} \\
\tau^*_{\lfloor \frac{n-1}{2^{i_2}} \rfloor} &= (Gparent(\lfloor \frac{n-1}{2^{i_2}} \rfloor)\ \ n-1) \quad \text{für} \quad i_2 = max\{i > i_1 \mid b_{i-1} \neq 0\} \\
\vdots \quad & \vdots \\
\tau^*_1 = \tau^*_{\lfloor \frac{n-1}{2^{j}} \rfloor} &= (Gparent(\lfloor \frac{n-1}{2^{j}} \rfloor)\ \ n-1) \quad \text{für} \quad j = min\{i \mid b_{i-1} \neq 0\}.
\end{aligned}
$$

Da $Gparent(\lfloor \frac{n-1}{2^j} \rfloor) = Gparent(1) = 0$ ist, folgt die Behauptung. □

Folgerung 5 *Für alle* $s \in SP = \{\lfloor \frac{n-1}{2^i} \rfloor \mid i \in \{0, \dots, \lfloor \log(n-1) \rfloor\}\}$ *definiere das* s*–te Endstück* $Heapi_s(n)$ *von* $Heapi(n)$ *durch*

$$
\begin{aligned}
& Heapi_s(n) := \tau^1_{Gparent(s),s} \circ \tau^1_{Gparent(s-1),s-1} \circ \dots \circ \\
& \tau^{b_{n,0}}_{Gparent(\lfloor \frac{s}{2^1} \rfloor),\lfloor \frac{n-1}{2^1} \rfloor} \circ \tau^1_{Gparent(\lfloor \frac{s}{2^1} \rfloor - 1),\lfloor \frac{s}{2^1} \rfloor - 1} \circ \dots \circ \\
& \vdots \\
& \tau^{b_{n,k_n-1}}_{0,1}.
\end{aligned}
$$

Dann gilt:

$$
Heapi_s(n-1) \circ (0\ \ s) = (Gparent(s)\ \ s) \circ (Heapi_s(n-1)). \qquad (10)
$$

Beweis: Da $s = \lfloor \frac{n-1}{2^i} \rfloor$ für ein $i \in \{0, \dots, \lfloor \log(n-1) \rfloor\}$ ist, überträgt sich die Argumentation des Hilfssatzes 14 direkt. □

Hilfssatz 15 *Sei* $n \in N$ *mit* $n - 1 = (b_{n,k_n} \dots b_{n,0})_2$ *und* $k_n = \lfloor \log(n-1) \rfloor$. *Genau dann, wenn* $\lfloor \frac{n-1}{2^i} \rfloor \neq \lfloor \frac{n-2}{2^i} \rfloor$ *gilt, ist* $b_{n,i-1} = 0$ *und* $b_{n-1,i-1} = 1$.

Beweis: *Dann:* Wenn $\lfloor \frac{n-1}{2^i} \rfloor \neq \lfloor \frac{n-2}{2^i} \rfloor$ gilt, dann ist $\lfloor \frac{n-1}{2^{i-1}} \rfloor$ linkes Kind von $\lfloor \frac{n-1}{2^i} \rfloor$ und $\lfloor \frac{n-2}{2^{i-2}} \rfloor$ rechtes Kind von $\lfloor \frac{n-2}{2^i} \rfloor$. Damit sind aber $b_{n,i-1} = 0$ und $b_{n-1,i-1} = 1$.
Genau dann: Wenn $b_{n,i-1} = 0$ und $b_{n-1,i-1} = 1$ sind, so gilt nach Hilfssatz 9 für alle $j < i$ $b_{n-1,j-1} \oplus b_{n,j-1} = 1$. Gleichzeitig ist $n - 1 = (b_{n,k_n} \dots b_{n,0})_2 = (b_{n-1,k_{n-1}} \dots b_{n-1,0})_2 + (1)_2 = n - 2 + 1$. Der Übertrag an der Stelle i entsteht somit, d.h $\lfloor \frac{n-1}{2^i} \rfloor = (b_{n,k_n} \dots b_{n,i})_2 \neq (b_{n-1,k_{n-1}} \dots b_{n-1,i})_2 = \lfloor \frac{n-2}{2^i} \rfloor$.□

Hilfssatz 16

$$Heapi(n-1)^{-1} \circ (Gparent(n-1)\ \ n-1) \circ Heapi(n) = (CaseB(n))^{-1}.$$

Beweis: Fallunterscheidung:

$n-1 \neq 2^k$ Es ist wieder $k = k_n = k_{n-1}$ und vereinfachenderweise $n-1 = (b_k \dots b_0)_2$ und $n-2 = (c_k \dots c_0)_2$.

Betrachte die Elemente der Höhe 0. Die Transposition $(Gparent(n-1)\ \ n-1)$ kann als eliminiert angesehen werden, da sie zweifach hintereinander auftaucht.

Für die Elemente $k \in \{n-2, \dots, \lfloor\frac{n-1}{2}\rfloor - 1\}$ gilt, daß $(Gparent(k-1)\ \ k-1)$ sowohl in $Heapi(n)$ als auch in $Heapi(n-1)$ auftaucht, so daß sie sich gegenseitig aufheben.

Betrachte nun die Elemente der Höhe 1. Wenn $b_0 \neq 0$ (impliziert $c_0 \neq 1$), so folgt nach Hilfssatz 15, daß $\lfloor\frac{n-1}{2}\rfloor = \lfloor\frac{n-2}{2}\rfloor$, d.h. die Transposition $(Gparent(\lfloor\frac{n-1}{2}\rfloor)\ \ \lfloor\frac{n-1}{2}\rfloor)$ und die Transposition $(Gparent(\lfloor\frac{n-2}{2}\rfloor)\ \ \lfloor\frac{n-2}{2}\rfloor)$ sind identisch. Sie tritt einmal in der Vielfachheit b_0 und einmal in der Vielfachheit c_0 also insgesamt in der Vielfachheit $b_0 \oplus c_0$ auf.

Wenn $b_0 = 0$ (also $c_0 = 1$), so folgt $\lfloor\frac{n-1}{2}\rfloor \neq \lfloor\frac{n-2}{2}\rfloor = \lfloor\frac{n-1}{2}\rfloor + 1$, d.h. die Transposition $(Gparent(\lfloor\frac{n-2}{2}\rfloor)\ \ \lfloor\frac{n-2}{2}\rfloor)$ und die Transposition $(Gparent(\lfloor\frac{n-1}{2}\rfloor + 1)\ \ \lfloor\frac{n-1}{2}\rfloor + 1)$ sind identisch. Sie tritt in der Vielfachheit $1 + c_0 = 2$ auf und kann eliminiert werden. Dagegen tritt

$(Gparent(\lfloor\frac{n-1}{2}\rfloor)\ \ \lfloor\frac{n-1}{2}\rfloor)$ mit der Vielfachheit $1 = b_0 \oplus c_0$ in $(Heapi(n-1))^{-1}$ auf, in $Heapi(n)$ hingegen nicht. Die Folgerung 5 zeigt auf, wie das Element $(Gparent(\lfloor\frac{n-1}{2}\rfloor)\ \ \lfloor\frac{n-1}{2}\rfloor)$ mit der Vielfachheit $1 = b_0 \oplus c_0$ zum Ende von $Heapi(n)$ getragen werden kann. Es bildet dort das letzte Element von $(CaseB(n))^{-1}$.

Für die nun folgenden Elemente $k \in \{\lfloor\frac{n-1}{2}\rfloor + 1, \dots, \lfloor\frac{n-1}{4}\rfloor - 1\}$ tauchen die Paare $(Gparent(k-1)\ \ k-1)$ wieder sowohl in $Heapi^*(n)$ als auch in $Heapi(n-1)$ auf. Sie können somit gestrichen werden.

Für die Höhe $i = 2, \dots, \lfloor\log(n-1)\rfloor$ folgt durch Kontraposition für $b_{i-1} \neq 0$ oder $c_{i-1} \neq 1$ nach Hilfssatz 15, daß $\lfloor\frac{n-1}{2^i}\rfloor = \lfloor\frac{n-2}{2^i}\rfloor$, d.h. die Transposition $(Gparent(\lfloor\frac{n-1}{2^i}\rfloor)\ \ \lfloor\frac{n-1}{2^i}\rfloor)$ und die Transposition

$(Gparent(\lfloor\frac{n-2}{2^i}\rfloor)\ \ \lfloor\frac{n-2}{2^i}\rfloor)$ sind identisch. Sie tritt einmal in der Vielfachheit b_i und einmal in der Vielfachheit c_i also insgesamt in der Vielfachheit $b_i \oplus c_i$ auf.

Falls $b_i = 0$ und $c_i = 1$ gilt, dann ist $(Gparent(\lfloor\frac{n-1}{2^i}\rfloor)\ \ \lfloor\frac{n-1}{2^i}\rfloor)$ in $Heapi(n-1)$ aber nicht in $Heapi(n)$, hat also die Vielfachheit $1 = b_i \oplus c_i$. Mittels der Folgerung 5 wird das Element $(Gparent(\lfloor\frac{n-1}{2}\rfloor)\ \ \lfloor\frac{n-1}{2}\rfloor)$ zum

Ende von $Heapi(n)$ getragen. Es bildet dort das i. letzte Element von $(CaseB(n))^{-1}$.

Da nun die Transpositionen $(Gparent(\lfloor\frac{n-2}{2^i}\rfloor)\ \ \lfloor\frac{n-2}{2^i}\rfloor)$ und

$(Gparent(\lfloor\frac{n-1}{2^i}\rfloor+1)\ \ \lfloor\frac{n-1}{2^i}\rfloor+1)$ identisch sind, werden sie sich eliminieren, was auch für

$(Gparent(k)\ \ k)$ gilt, wobei k ein Element bis zum Erreichen der nächsten Höhe ist.

$n-1=2^k=(1\ \ 0\ldots 0)_2$. Es ist $k_n=k_{n-1}+1$, also $n-1=(b_k\ldots b_0)_2$ und $n-2=(c_{k-1}\ldots c_0)_2$. Die Argumentation läuft jedoch analog:

Es gilt für alle $i\in\{0,\ldots,k_n-1\}$: $b_{n,i}=0$ und $b_{n-1,i}=1$. Deshalb ist $(Gparent(\lfloor\frac{n-1}{2^i}\rfloor)\ \ \lfloor\frac{n-1}{2^i}\rfloor)$ in $Heapi(n-1)$, aber nicht in $Heapi(n)$, hat also die Vielfachheit $1=b_i\oplus c_i$. Mittels der Folgerung 5 wird das Element $(Gparent(\lfloor\frac{n-1}{2}\rfloor)\ \ \lfloor\frac{n-1}{2}\rfloor)$ zum Ende von $Heapi(n)$ getragen. Es bildet dort das i.-letzte Element von $(CaseB(n))^{-1}$.

Da nun die Transpositionen $(Gparent(\lfloor\frac{n-2}{2^i}\rfloor)\ \ \lfloor\frac{n-2}{2^i}\rfloor)$ und

$(Gparent(\lfloor\frac{n-1}{2^i}\rfloor+1)\ \ \lfloor\frac{n-1}{2^i}\rfloor+1)$ identisch sind, werden sie sich eliminieren, was auch für

$(Gparent(k)\ \ k)$ gilt, falls k ein Element bis zum Erreichen der nächsten Höhe ist. □

Hilfssatz 17 *(Schlüssellemma)*

$$Heapi(n)\circ Swap(0,n-1)\circ CaseB(n)\circ Heapi(n-1)^{-1}=id_{\{0,\ldots,n-1\}}. \quad (11)$$

Beweis: Die Gleichung

$$Heapi(n)\circ Swap(0,n-1)\circ CaseB(n)\circ Heapi(n-1)^{-1}=id_{\{0,\ldots,n-1\}}$$

ist mittels Rechts- und Linksmultiplikation mit $Heapi(n-1)$ bzw. mit $Heapi(n-1)^{-1}$ umformbar in:

$$Heapi(n-1)^{-1}\circ Heapi(n)\circ Swap(0,n-1)\circ CaseB(n)=id_{\{0,\ldots,n-1\}}.$$

Damit folgt aus Hilfssatz 14 die Äquivalenz mit:

$$Heapi(n-1)^{-1}\circ(Gparent(n-1)\ \ n-1)\circ Heapi(n)\circ CaseB(n)=id_{\{0,\ldots,n-1\}}.$$

Letztendlich mit Hilfssatz 16:

$$(CaseB(n))^{-1}\circ CaseB(n)=id_{\{0,\ldots,n-1\}}.\square$$

Beispiel 1 *Sei $n=15$, also $n-1=14=(b_3\ \ b_2\ \ b_1\ \ b_0)_2=(1\ \ 1\ \ 1\ \ 0)_2$ und $n-2=13=(c_3\ \ c_2\ \ c_1\ \ c_0)_2=(1\ \ 1\ \ 0\ \ 1)_2$. Die Transposition von den Stellen i und j $(\tau_{i,j})$ sei durch die Zykelschreibweise $(i\ \ j)$ beschrieben.*

$Heapi(15) = (14\ \ 3)^1(13\ \ 6)^1(12\ \ 1)^1(11\ \ 5)^1(10\ \ 2)^1(9\ \ 4)^1(8\ \ 0)^1(7\ \ 3)^0$
$(6\ \ 1)^1(5\ \ 2)^1(4\ \ 0)^1(3\ \ 1)^1(2\ \ 0)^1(1\ \ 0)^1$,
$Swap(0,14) = (0\ \ 14)^1$,
$CaseB(15) = (0\ \ 7)^1(0\ \ 3)^1(0\ \ 1)^0$,
$Heapi(14)^{-1} = (1\ \ 0)^1(2\ \ 0)^1(3\ \ 1)^0(4\ \ 0)^1(5\ \ 2)^1(6\ \ 1)^1(7\ \ 3)^1(8\ \ 0)^1(9\ \ 4)^1$
$(10\ \ 2)^1(11\ \ 5)^1(12\ \ 1)^1(13\ \ 6)^1$,
also

$$\begin{array}{l}
(1\ \ 0)(2\ \ 0)(4\ \ 0)(5\ \ 2)(6\ \ 1)(7\ \ 3)(8\ \ 0)(9\ \ 4)(10\ \ 2)(11\ \ 5)(12\ \ 1)(13\ \ 6)\\
(14\ \ 3)(13\ \ 6)(12\ \ 1)(11\ \ 5)(10\ \ 2)(9\ \ 4)(8\ \ 0)(6\ \ 1)(5\ \ 2)(4\ \ 0)(3\ \ 1)(2\ \ 0)(1\ \ 0)\\
(0\ \ 14)\\
(0\ \ 7)(0\ \ 3) =\\
(1\ \ 0)(2\ \ 0)(4\ \ 0)(5\ \ 2)(6\ \ 1)(7\ \ 3)(6\ \ 1)\\
(5\ \ 2)(4\ \ 0)(3\ \ 1)(2\ \ 0)(1\ \ 0)\\
(0\ \ 7)(0\ \ 3) =\\
(1\ \ 0)(2\ \ 0)\\
(3\ \ 1)(2\ \ 0)(1\ \ 0)(0\ \ 3) =\\
id_{\{0,\ldots,14\}}.
\end{array}$$

Satz 11 *Der best-case von WEAK-HEAPSORT wird bei aufsteigender Vorsortierung der Elemente ($a[i] < a[i+1]$ für $i \in \{0,\ldots,n-2\}$) erzielt.*

Beweis: Ohne Beschränkung der Allgemeinheit sei $a[i] = i$ für alle $i \in \{0,\ldots,n-1\}$. Der Satz wird mittels vollständiger Induktion über die Anzahl der Elemente bewiesen. Induktionsanfang (n=2): Die Eingabe für WEAK-HEAPSORT ist $a[0] = 0$ und $a[1] = 1$, $r[0] = 0$ und $r[1] = 0$. Nach der Aufbauphase (es tritt genau eine Vertauschung ein) gilt: $a[0] = 1$ und $a[1] = 0$, $r[0] = 0$ und $r[1] = 1$. Dann wird $a[2]$ auf den Wert von $a[0]$ gesetzt und das Array ist sortiert. Die Vergleichsanzahl ist 1 und beschreibt den best-case.

Induktionsschritt($n-1 \Longrightarrow n$):

Das Schlüssellemma

$$Heapi(n) \circ Swap(0,n-1) \circ CaseB(n) \circ Heapi(n-1)^{-1} = id_{\{0,\ldots,n-1\}}$$

beschreibt in Verbindung mit Satz 10 den richtig beschriebenen Übergang von einem vorsortierten Array der Länge n hin zu einem vorsortierten Array der Länge $n-1$.

Nach Induktionsvoraussetzung tritt für ein vorsortiertes Array der Länge $n-1$ der best-case ein. Der Fall b) impliziert dann die minimale Anzahl $\lfloor\log(n+1)\rfloor - 1$ von Vergleichen. Somit tritt auch für das vorsortierte Array der Länge n der best-case ein. □

6 Die worst-case Analysen

Krise ist ein produktiver Zustand. Man muß ihr nur den Beigeschmack der Katastrophe nehmen.

Max Frisch

6.1 Die worst-case Analyse von HEAPSORT

Das Negative interessiert mehr als das Positive. Das hat Shakespeare schon gewußt. Im Grunde ist das erfreulich, denn es beweist, daß das Negative immer noch die Ausnahme ist.

Tennessee Williams

Es wird kurz an die klassische Version von HEAPSORT nach Williams(1964) bzw. Floyd(1964)) erinnert.
Seien mit $a[1], \ldots, a[n]$ die Arrayinhalte für eine zu sortierende Objektmenge der Größe n bezeichnet. Die Heapeigenschaft gilt für die Position i als erfüllt, wenn $a[i] \leq a[2i]$ oder $i > \lfloor n/2 \rfloor$ ist und $a[i] \leq a[2i+1]$ oder $i \geq \lfloor n/2 \rfloor$ ist. Ein Array wird als Heap bezeichnet, wenn die Heapeigenschaft für alle Positionen $i \in \{1, \ldots, n\}$ erfüllt ist. Diese Darstellung entspricht einem eingebetteten Binärbaum, bei dem die Werte an jedem Knoten i kleiner sind als die der zugehörigen Kinder.
Die Prozedur $ReHeap(m,i)$ betrachtet nur die Arraypositionen von $1, \ldots, m$, die in dem Teilbaum mit Wurzel i liegen. Ist die Heapeigenschaft nur an der Wurzel i verletzt, so wandelt $ReHeap(m,i)$ den durch i beschriebenen Teilbaum in einen Heap. Dies führt zu dem bekannten HEAPSORT Rahmenprogramm:

```
PROCEDURE HeapSort
  FOR i = n DIV 2 DOWNTO 1 DO ReHeap(n,i) OD (*Generierungphase*)
  FOR m = n DOWNTO 2 DO                      (*Auswahlphase*)
    Swap(a[1],a[m])
    IF m <> 2 THEN ReHeap(m-1,1)
  OD
END HeapSort.
```

Die Generierungsphase wird auch Aufbau–, Heapify–, oder Heap–Creation–Phase genannt, die Auswahlphase auch häufig Selection– oder Sift-Up Phase. Die Prozedur *ReHeap* kann rekursiv gestaltet werden:

```
PROCEDURE ReHeap(m,i)
  IF 2i <= m THEN
     IF 2i = m
```

```
    THEN IF a[i] > a[2i] THEN Swap(a[i],a[2i]) FI
    ELSE IF a[i] <= a[2i+1] THEN l = 2i ELSE l = 2i+1 FI
         IF a[i] > a[l] THEN
            Swap(a[i],a[l])
            ReHeap(l,m)
         FI
    FI
  FI
END ReHeap.
```

Pro Rekursionsstufe werden max. zwei Vergleiche benötigt. Die Rekursion terminiert in drei Fällen: Im Fall eines Blattes ($2i > m$) erfolgt kein Vergleich, im Fall eines vereinzelten linken Kindes ($2i = m$) wird ein Vergleich benötigt und im Fall, daß die Heapeigenschaft für den betrachteten Knoten i erfüllt wird ($a[i] \leq a[l]$), sind zwei Vergleiche durchgeführt worden. Bezeichne mit $path[1], \ldots, path[r]$ die Objekte auf dem in *ReHeap* generierten Pfad der Länge r, so gilt: $a[path[r]] > a[path[r-1]]$ und $a[path[r]] \leq a[2path[r]]$ als auch $a[path[r]] \leq a[2path[r]+1]$. Die Länge dieses Pfades ist durch die Tiefe d des durch i beschriebenen Teilbaumes beschränkt. Es werden dementsprechend max. $2d$ wesentliche Vergleiche pro *ReHeap*–Aufruf benötigt. Demnach gilt es, die Summe der betrachteten Baumtiefen zu studieren.

Hilfssatz 18 *Die Summe der Tiefen von den betrachteten Heaps während der Generierungsphase ist höchstens $n - 1$.*

Beweis: (Wegener (1995)) Es sind nur die Knoten $1, \ldots, \lfloor n2^{-d} \rfloor$ Wurzeln von Bäumen, deren Tiefe mindestens d beträgt. Indem $\lfloor n2^{-d} \rfloor$ für $d \in \{1, \ldots \lfloor \log n \rfloor\}$ aufsummiert wird, werden die Bäume der Tiefe d genau d–mal gezählt:

$$\sum_{d=1}^{\lfloor \log n \rfloor} \lfloor n2^{-d} \rfloor < \sum_{d=1}^{\infty} n2^{-d} = n. \square$$

Hilfssatz 19 *Die Summe der Tiefen der Heaps während der Auswahlphase ist für $n \geq 1$:*

$$n\lfloor \log n \rfloor - 2 \cdot 2^{\lfloor \log n \rfloor} + \lfloor \log 2 \rfloor + 2$$

bzw. $n \log n - c(n)n + \lfloor \log 2 \rfloor + 2$ *mit* $min\{c(n) \mid n \in N\} \approx 1.91393$.

Beweis: (Wegener (1995)) Es gilt, die Heaps der Größe $n, \ldots, 2$ zu betrachten. Die Tiefe des Heaps mit Wurzel i ist $\lfloor \log i \rfloor$. Die Summe von $\lfloor \log i \rfloor$ über alle $i \in \{2, \ldots, n\}$ wird in Analogie zu Hilfssatz 3 aufgeteilt:

$$\sum_{i=2}^{n} \lfloor \log i \rfloor \quad = \quad \sum_{i=1}^{n} \lfloor \log i \rfloor$$

$$\begin{aligned}
&= \sum_{i=1}^{\lfloor \log n \rfloor - 1} i2^i + \lfloor \log n \rfloor (n - 2^{\lfloor \log n \rfloor} + 1) \\
&\overset{HS2}{=} (\lfloor \log n \rfloor - 2)2^{\lfloor \log n \rfloor} + 2 + \lfloor \log n \rfloor (n - 2^{\lfloor \log n \rfloor} + 1) \\
&= n\lfloor \log n \rfloor - 2 \cdot 2^{\lfloor \log n \rfloor} + \lfloor \log n \rfloor + 2.
\end{aligned}$$

Es soll $n\lfloor \log n \rfloor - 2 \cdot 2^{\lfloor \log n \rfloor}$ in einen Ausdruck der Art $n \log n - c(n)n$ geformt werden, d.h. es gilt das Verhalten von

$$c(n) = (\log n - \lfloor \log n \rfloor) - \frac{2 \cdot 2^{\lfloor \log n \rfloor}}{n} \tag{12}$$

zu studieren. Für alle $x \in R$, $x \in I_k = [2^k, 2^{k+1} - 1]$, gilt: $\lfloor \log n \rfloor = k$. Sei f eine reellwertige Funktion mit

$$f(x) = \log x - k + \frac{2^{k+1}}{x}.$$

Dann gilt:

$$f'(x) = \frac{1}{\ln 2x} - \frac{2^{k+1}}{x^2} = 0 \Longleftrightarrow x = (2 \ln 2)2^k.$$

Weiterhin gilt für $x \in I_k$: $f''(x) > 0$. Demnach nimmt $c(n)$ für $n \approx (2 \ln 2)2^k$ den Minimalwert $c(n) \approx 1.91393$ an.□

Satz 12 *Die worst-case Zahl wesentlicher Vergleiche von HEAPSORT ist durch* $2n \log n + (2 - 2c(n))n + 2\lfloor \log 2 \rfloor$ *beschränkt. Für den Aufbau eines Heaps genügen* $2n - 2$ *wesentliche Vergleiche.* □

6.2 Die worst-case Analyse von BOTTOM–UP–HEAPSORT

Schlechte Burschen zu entlarven, ist ein gutes, ein verdienstvolles Werk. (La Roche)

Friedrich von Schiller
Der Parasit, I, 2

Die Vergleichsanzahl von HEAPSORT unterscheidet sich von der unteren Schranke im wesentlichen um den Faktor 2. Dieser Faktor ergibt sich dadurch, daß in der *ReHeap*–Prozedur pro Knoten des speziellen Weges jeweils ein Minimum dreier Objekte zu bilden ist.
Die Idee der HEAPSORT–Variante BOTTOM-UP-HEAPSORT nach Carlsson(1987b) und Wegener(1993) beinhaltet die Entkoppelung der dazu hinreichenden und notwendigen 2 Vergleiche. Im sukzessiven Vergleich zwischen den beiden jeweils betrachteten Kindern wird der spezielle Pfad bis hin zu einem

Blatt gesucht. Und erst in einem zweiten Schritt wird die Stelle bestimmt, bis zu der das neue Wurzelelement zur Wiederherstellung der Heapanforderung sinken soll. Die Suche wird bottom–up organisiert, also von dem gefundenen Blatt zur Wurzel hin. Dadurch wird der Beobachtung Rechnung getragen, daß die erwartete Tiefe des einsinkenden Elementes groß ist.
Es ergibt sich demnach folgende Struktur der BOTTOM–UP–Ansatzes:

```
PROCEDURE BottomUpReHeap(m,i)
  LeafSearch(m,i)
  BottomUpSearch(i,j)
  Interchange(i,j)
END BottomUpReHeap.
```

Dabei ist die Variable j global festgelegt. Seien mit $a[1], \ldots, a[n]$ die Arrayinhalte für eine zu sortierende Objektmenge der Größe n bezeichnet und mit m die betrachtete Obergrenze des aktuell gültigen Heaps. Das letzte Element des von i aus startenden speziellen Weges wird nun wie folgt bestimmt:

```
PROCEDURE LeafSearch(m,i)
  j = i
  WHILE 2j < m DO
    IF a[2j] < a[2j + 1] THEN j = 2j
                         ELSE j = 2j + 1
    FI
  OD
  IF 2j = m THEN j=m
END LeafSearch.
```

Es ist festzuhalten, daß im Falle eines vereinzelten linken Kindes ($2j = m$) ein Vergleich eingespart wird. Bezeichne mit $b[1], \ldots, b[r]$ die Objekte auf dem speziellen Pfad der Länge r, zusätzlich sei $b[0] = -\infty$ und $b[r+1] = \infty$. Wenn mit x das Wurzelobjekt festgehalten wird, soll nun die Stelle l gefunden werden, so daß $b[l] \leq x < b[l+1]$ gilt (Zum Vergleich: Die alte *ReHeap*–Prozedur bestimmte ein l mit $b[l] < x \leq b[l+1]$):

```
PROCEDURE BottomUpSearch(i,j)
  WHILE ( (i < j) AND (a[i] < a[j]) ) DO j = j DIV 2 OD
END BottomUpSearch.
```

Die erste Bedingung verhindert einen Vergleich von $a[i]$ mit sich selbst. Der nun noch durchzuführende Datentransport kann effizient mittels Bitoperationen gestaltet werden:

```
PROCEDURE Interchange(i,j)
  l = bin(j)-bin(i)
```

```
  x = a[i]
  FOR k = l-1 DOWNTO 0 DO a[j DIV 2^(k+1)] = a[j DIV 2^k] OD
  a[j] = x
END Interchange.
```

Falls für $i \neq j$ $a[i] \neq a[j]$ ist, so produziert $BottomUpReHeap(m, i)$ das gleiche Ergebnis wie $ReHeap(m, i)$. Die Korrektheit des Ansatzes folgt somit aus der von HEAPSORT.

Hilfssatz 20 *Die Anzahl der Vergleiche während der Aufrufe von $LeafSearch$ in der Generierungsphase ist mindestens $n - \lceil \log(n+1) \rceil - \lceil \log n \rceil + 1$ und höchstens $n - 2$.*

Beweis: (Wegener (1993)[7]) Für $n = 2^k - 1$ ist der eingebettete Baum vollständig. Es befinden sich auf der Höhe $l \in \{1, \ldots, k-1\}$ genau $2^{k-(l+1)}$ Knoten. Jeder dieser Knoten verursacht l Vergleiche. Damit ergibt sich die Gesamtzahl von:

$$\begin{aligned}
\sum_{l=1}^{k-1} l 2^{k-l-1} &= \sum_{m=1}^{k-1} (k-m) 2^{m-1} \\
&= \sum_{m=1}^{k-1} k 2^{m-1} - \sum_{m=1}^{k-1} m 2^{m-1} \\
&= k \left(\sum_{m=0}^{k-2} 2^m \right) - \frac{1}{2} \left(\sum_{m=0}^{k-1} m 2^m \right) \\
&\overset{HS2}{=} k(2^{k-1} - 1) - \frac{1}{2}((k-2)2^k + 2) \\
&= 2^k - k - 1
\end{aligned}$$

Vergleichen. Nun wird der nächste Level von links nach rechts aufgefüllt, d.h. $n \in \{2^k, \ldots, 2^{k+1} - 1\}$. Somit ist $i = n - (2^k - 1)$ die Anzahl der Knoten auf dem untersten Level k. Im Bezug auf den vollständigen Baum $(2^k - 1)$ erhöht sich die Höhe von allen Vorfahren der durch i beschriebenen neuen Blätter um 1. Ist i ungerade, d.h. beschreibt i ein geschwisterloses Blatt, so ist kein zusätzlicher Vergleich an der Wurzel eines Vorfahrens nötig. Somit ergeben die akkumulierte Anzahl der Vorfahren zu den Knoten 2^k bis $n - 1$ eine obere Schranke für die Anzahl der zusätzlichen Vergleiche:

$$\begin{aligned}
\sum_{r=1}^{k} \lceil \frac{i-1}{2^r} \rceil &< k + \sum_{i=1}^{k} \frac{i-1}{2^r} \\
&\leq k + (i-1) \cdot \sum_{i=1}^{\infty} \frac{1}{2^r} \\
&= i - 1 + k.
\end{aligned}$$

[7] Ebd. wird die untere Grenze verkürzt bewiesen.

Aufgrund der Ganzzahligkeit der linken und rechten Seite gilt: $\sum_{r=1}^{k} \lceil \frac{i-1}{2^r} \rceil \leq i - 2 + k$. Demnach ist der worst-case der durch *LeafSearch* bedingten Vergleichsanzahl für die Aufbauphase durch $2^k - 1 - k + i - 2 + k = n - 2$ nach oben beschränkt.
Die obere Schranke ist aber auch immer größer gleich $2^k - 1 - k + (i-1) = n - (k+1) = n - \lceil \log(n+1) \rceil$, da für die Anzahl der Vergleiche auf der untersten Ebene die folgende Ungleichung gilt:

$$\begin{aligned} \sum_{r=1}^{k} \lceil \frac{i-1}{2^r} \rceil &\geq \lceil \sum_{r=1}^{k} \frac{i-1}{2^r} \rceil \\ &= \lceil i - 1 - \frac{i-1}{2^k} \rceil \\ &= i - 1. \end{aligned}$$

Nur wenn die Prozedur *LeafSearch* mit einer Wurzel des speziellen Pfades aufgerufen wird, läßt sich im best-case noch ein Vergleich sparen. Für $n \neq 2^k$ liegen nicht mehr als $\lfloor \log n \rfloor - 1$ innere Knoten auf dem speziellen Pfad. □

Hilfssatz 21 *Die Anzahl der Vergleiche während der Aufrufe von LeafSearch in der Auswahlphase ist für $n \geq 3$ höchstens:*

$$n\lfloor \log(n-2) \rfloor - 2 \cdot 2^{\lfloor \log(n-2) \rfloor} \lfloor \log(n-2) \rfloor + 2$$

bzw. $n \log n - c(n)n$ *mit* $min\{c(n) \mid n \in N\} \approx 1.91393$.

Beweis:(Wegener (1993)) Die Prozedur *LeafSearch* wird in der Auswahlphase für die Parameter $(i, 1)$, $i \in \{n-1, \ldots, 2\}$, aufgerufen. Für jedes i werden maximal $\lfloor \log(i-1) \rfloor$ Vergleiche durchgeführt. Die Summe von $\lfloor \log(i-1) \rfloor$ über alle $i \in \{2, \ldots, n\}$ wird in Analogie zu Hilfsatz 19 zum behaupteten Ergebnis ausgewertet:

$$\begin{aligned} & \sum_{i=2}^{n-1} \lfloor \log(i-1) \rfloor \\ = \; & \sum_{i=1}^{n-2} \lfloor \log i \rfloor \\ = \; & \sum_{i=1}^{\lfloor \log(n-2) \rfloor - 1} i 2^i + \lfloor \log(n-2) \rfloor (n - 2^{\lfloor \log(n-2) \rfloor} - 1) \\ \overset{HS2}{=} \; & (\lfloor \log(n-2) \rfloor - 2) 2^{\lfloor \log(n-2) \rfloor} + 2 + \lfloor \log(n-2) \rfloor (n - 2^{\lfloor \log(n-2) \rfloor} - 1) \\ = \; & n \lfloor \log(n-2) \rfloor - 2 \cdot 2^{\lfloor \log(n-2) \rfloor} - \lfloor \log(n-2) \rfloor + 2. \end{aligned}$$

Es soll $n\lfloor \log(n-2) \rfloor - 2 \cdot 2^{\lfloor \log(n-2) \rfloor}$ in einen Ausdruck der Art $n \log n - c(n)n$ geformt werden, d.h. es gilt das Verhalten von

$$c(n) = (\log n - \lfloor \log(n-2) \rfloor) - \frac{2 \cdot 2^{\lfloor \log(n-2) \rfloor}}{n} \tag{13}$$

zu studieren. Für alle $x \in R$, $x \in I_k = [2^k + 2, 2^{k+1} + 1]$, gilt: $\lfloor \log(n-2) \rfloor = k$. Sei f eine reellwertige Funktion mit

$$f(x) = \log x - k + \frac{2^{k+1}}{x}.$$

Dann gilt:

$$f'(x) = \frac{1}{\ln 2x} - \frac{2^{k+1}}{x^2} = 0 \Longleftrightarrow x = (2 \ln 2) 2^k.$$

Weiterhin gilt für $x \in I_k$: $f''(x) > 0$. Demnach nimmt $c(n)$ für $n \approx (2 \ln 2) 2^k$ den Minimalwert $c(n) \approx 1.91393$ an.□

Hilfssatz 22

$$s_n = \sum_{i=0}^{n} \frac{i}{2^i} = 2 \left(1 - \frac{n+2}{2^{n+1}} \right).$$

Beweis: (Verteilungs–, oder Gleichsetzungsmethode) Einerseits ist:

$$\begin{aligned} s_{n+1} &= \sum_{i=1}^{n+1} \frac{i}{2^i} = \sum_{i=0}^{n} \frac{n+1}{2^{i+1}} = \frac{1}{2} s_n + \frac{1}{2} \cdot \sum_{i=0}^{n} \frac{1}{2^i} \\ &= \frac{1}{2} s_n + 1 - \frac{1}{2^{n+1}} \end{aligned}$$

und andererseits gilt $s_{n+1} = (n+1)/2^{n+1} + s_n$. Gleichsetzung und Auflösen nach s_n liefert die Behauptung. □

Satz 13 *BOTTOM–UP–HEAPSORT benötigt nicht mehr als* $1.5 n \log n + (2 - c(n)) n + O(\log^2 n)$ *Vergleiche.*

Beweis: (Wegener (1993)[8]) Da in der Prozedur *BottomUpSearch* nicht mehr Vergleiche gemacht werden als während des entsprechenden Aufrufes von *LeafSearch*, ist nach Hilfssatz 20 die Anzahl der Vergleiche in der Generierungsphase durch $2n - 4$ beschränkt. Hilfssatz 21 garantiert, daß die Anzahl der Vergleiche in der Auswahlphase, die durch die Aufrufe von *LeafSearch* verursacht werden, durch $n \log n - c(n) n$ begrenzt ist.
Es gilt demnach noch, die Anzahl der Vergleiche für die *BottomUpSearch*–Aufrufe zu untersuchen. Es wird gezeigt, daß diese Zahl durch $0.5 n \log n + 2n + O(\log^2 n)$ begrenzt ist.
Ohne Beschränkung der Allgemeinheit wird angenommen, daß der Heap die Elemente von 1 bis n enthält.
Sei $n = 2^k - 1 + \alpha 2^k$ für ein $\alpha \in [0, 1)$ gegeben. Dabei bezeichnet α den Füllgrad des untersten Levels, d.h die Level $0, \ldots, k-1$ sind vollständig und der Level

[8] Ebd. wird das etwas schwächere Ergebnis von $1.5 n \log n + (2.25 - c(n)) n + O(log^2 n)$ gezeigt.

k mit $\alpha 2^k$ Knoten gefüllt. Es gibt $\lceil n/2 \rceil$ Blätter. Die Elemente von 1 bis $\lceil n/2 \rceil$ werden als *klein* und die anderen Elemente als *groß* bezeichnet. Nach den ersten $\lceil n/2 \rceil$ *BottomUpSearch*–Aufrufen sind nur noch die große Elemente im Heap. Die Heapeigenschaft impliziert, daß die Nachfolger von großen Elementen groß und die Vorgänger von kleinen Elementen klein sein müssen. Damit sind maximal $\lceil n/4 \rceil$ Blätter klein und demnach auch die einzusinkenden Wurzeln.
Die Idee gestaltet sich nun wie folgt: Die bottom-up betrachteten Wege von kleinen Wurzelelementen werden durch die aktuelle Tiefe $d(t)$ abgeschätzt. Sie dominieren insgesamt die verbleibende Laufzeit. Für die verbleibenden $\lceil n/2 \rceil - \lceil n/4 \rceil$ großen Elemente wird gezeigt, daß deren mittlere Tiefe sehr groß und damit der Bottom–Up–Pfad kurz sein muß. Induktiv werden dann die folgenden Höhen in groß und klein partitioniert und zu einem Gesamtergebnis verbunden. Die Summe der insgesamt $\lceil n/2 \rceil$ Tiefenwerte $d(t)$ beträgt:

$$\alpha 2^k k + \left(\lceil \frac{n}{2} \rceil - \alpha 2^k\right)(k-1) = \lceil \frac{n}{2} \rceil k - \left(\lceil \frac{n}{2} \rceil - \alpha 2^k\right). \tag{14}$$

Betrachte den m-ten *BottomUpSearch*–Aufruf eines großen Elementes. Sei d_m die Tiefe im aktuellen Baum, an dem das Wurzelelement für diesen Aufruf letztendlich landet. Damit ergeben sich unmittelbar $d(t) - d_m + 1$ Vergleiche in der Rückwärtssuche. Große Elemente steigen nur dann auf, wenn sie mit großen Elementen verglichen werden, demnach kann die Summe der d_m nicht größer als die minimale Summe von Tiefen eines aus $\lceil n/2 \rceil - \lceil n/4 \rceil$ Knoten bestehenden Binärbaumes sein. In diesem Fall sind die Level $0, \ldots, k-3$ vollständig und der Level $k-2$ bis zur Position $\lfloor \alpha 2^{k-2} \rfloor$ aufgefüllt. Die Summe der Tiefen beträgt demnach:

$$\begin{aligned}
\sum_{i=0}^{k-3} i 2^i + \lfloor \alpha 2^{k-2} \rfloor (k-2) &\overset{HS2}{=} (k-4)2^{k-2} + 2 + \alpha 2^{k-2}(k-2) \\
&\geq (k-4)\frac{n}{4} + (k-4)\frac{1}{4} - (k-4)\alpha 2^{k-2} + 2 + \\
&\quad (\alpha 2^{k-2} - 1)(k-2) \\
&\geq (k-4)\frac{n}{4} + 2\alpha 2^{k-2} - k + 3.
\end{aligned}$$

Zusammenfassend ergeben sich für die ersten $\lfloor n/2 \rfloor$ Aufrufe von *BottomUpSearch* maximal

$$\lceil \frac{n}{2} \rceil k - \left(\lceil \frac{n}{2} \rceil - \alpha 2^k\right) - (k-4)\frac{n}{4} - 2\alpha 2^{k-2} + k - 3 + \lceil \frac{n}{2} \rceil - \lceil \frac{n}{4} \rceil$$

$$\leq \left(\lceil \frac{n}{2} \rceil - \frac{n}{4}\right) k + \frac{3n}{4} + \alpha 2^{k-1} + k$$

Vergleiche.

Letztendlich beträgt die Anzahl der Vergleiche in der Auswahlphase also weniger als:

$$
\begin{aligned}
& \sum_{i=0}^{k-1}\left(\lceil \frac{n}{2^{i+1}} \rceil - \frac{n}{2^{i+2}}\right)(k-i) + \frac{3n}{2^{i+2}} + \alpha 2^{k-i-1} + k - i \\
\leq \quad & \sum_{i=0}^{k-1}\left(\frac{n}{2^{i+2}} + 1\right)(k-i) + 3n\sum_{i=0}^{k-1}\frac{1}{2^{i+2}} + \alpha\sum_{i=1}^{k} 2^{i-1} + \sum_{i=0}^{k-1}(k-i) \\
\leq \quad & \sum_{i=0}^{k-1}\left(\frac{n}{2^{i+2}}\right)(k-i) + \frac{3n}{2} + \alpha 2^k + 2\sum_{i=0}^{k-1}(k-i) \\
\leq \quad & \frac{n}{4}\left(k\sum_{i=0}^{k-1}\frac{1}{2^i} - \sum_{i=0}^{k-1}\frac{i}{2^i}\right) + \frac{3n}{2} + \frac{n}{2} + 2\sum_{i=1}^{k} i \\
\overset{HS22}{\leq} \quad & \frac{nk}{2} - \frac{n}{2} + \frac{n}{4}\cdot\frac{k+2}{2^k} + 2n + k^2 \\
\leq \quad & \frac{nk}{2} - \frac{n}{2} + O(k) + 2n + k^2.
\end{aligned}
$$

Mit $k = \lceil \log n \rceil < \log n + 1$ ergibt sich die zu beweisende Grenze von $0.5n\log n + 2n + O(log^2 n)$. □
Fleischer (1991) sowie Schaffer und Sedgewick (1993) haben worst-case Beispiele angegeben, bei denen die Anzahl der wesentlichen Vergleiche für BOTTOM-UP-HEAPSORT gleich $1.5n\log n - o(n\log n)$ ist.

6.3 Die worst-case Analyse von MDR–HEAPSORT

Schlaf ist mir lieb, doch über alles preise ich, Stein zu sein. Währt Schande und Zerstören, nenn ich es Glück: Nicht sehen und nicht hören.

Michelangelo Buonarotti

McDiarmid and Reeds (1989) Variante des BOTTOM-UP-HEAPSORT, kurz MDR–HEAPSORT ermöglicht es, weitere Vergleiche einzusparen, indem alte Informationen verwendet werden. Es wird allerdings ein zusätzliches Array $info$ benötigt, daß diese alten Informationen verwaltet. Die möglichen Belegungen von $info[j]$ sind *unbekannt*, *links* und *rechts* (abgekürzt mit u, l und r) und können mit zwei Bits codiert werden. Ist $info[j] = l$, dann enthält das linke Kind ein kleineres Element als das rechte, im Falle $info[j] = r$ ist es entsprechend genau umgekeht. Wenn $info[j] = u$ gilt, ist keine Dominanz der Kinder untereinander bekannt. Da die Information $info[j]$ nur für die Elemente $1, \ldots, \lfloor (n-1)/2 \rfloor$ das Einsparen von Vergleichen verspricht, werden insgesamt nur $2\lfloor (n-1)/2 \rfloor < n$ Zusatzbits benötigt. Somit ist MDR-HEAPSORT als ein

direkter Konkurent zu WEAK-HEAPSORT anzusehen, da sich die Zusatzinformation pro Knoten auch auf ein Bit beschränkt.
Es sei $info[j]$ mit u initialisiert. Die Veränderungen zu BOTTOM-UP-HEAPSORT sind gering:

```
PROCEDURE MDRReHeap(m,i)
  LeafSearch(m,i)
  BottomUpSearch(i,j)
  Interchange(i,j)
END MDRReHeap.
```

mit

```
PROCEDURE LeafSearch(m,i)
  j = i
  WHILE 2j < m DO
     IF info[j] = l THEN j = 2j
     ELSEIF info[j] = r THEN j  = 2j + 1
     ELSEIF a[2j] < a[2j + 1] THEN info[j] = l
                                   j = 2j
     ELSE info[j] = r
          j = 2j + 1
    FI
  OD
  IF 2j = m THEN j=m
END LeafSearch.
```

Die Prozedur $BottomUpSearch(i, j)$ wird nicht modifiziert und der Informationsverlust, der durch das Einsinken des Wurzelelementes eintritt, wird wie folgt verwaltet:

```
PROCEDURE Interchange(i,j)
  l = bin(j)-bin(i)
  x = a[i]
  FOR k = l-1 DOWNTO 0 DO a[j DIV 2^(k+1)] = a[j DIV 2^k]
                          info[j DIV 2^(k+1)] = u
                       OD
  a[j] = x
END Interchange.
```

Die Korrektheit des Algorithmus folgt direkt aus der Korrektheit von BOTTOM-UP-HEAPSORT. Auch MDR-HEAPSORT konstruiert bei paarweise verschiedenen Elementen in jedem $MDRReHeap$–Schritt den gleichen Heap wie das bekannte HEAPSORT–$ReHeap$.
Im folgenden soll die worst-case Analyse von Wegener (1992) dargestellt werden.

Um die Anzahl der Vergleiche besser studieren zu können, wird die Bezeichnung $Kiesel$ (pebble) eingeführt. Ein Knoten j besitzt einen Kiesel, wenn $info[j] \neq u$ und j zwei Kinder im aktuellen Heap hat. Kiesel werden nur in der Prozedur $LeafSearch$ *erzeugt*. Sie *verschwinden*, wenn ein Knoten in der Auswahlphase das rechte Kind einbüßt und sie werden in der Prozedur $Interchange$ *entfernt*. Während eines Aufrufes von $MDRReHeap$ werden zuerst alle Knoten des speziellen Pfades (SP) $b[1], \ldots, b[d]$ mit einem Kiesel versehen und dann für ein spezielles Anfangsstück $b[1], \ldots, b[l]$ wieder entfernt. Dabei besitzt $b[d]$ keinen Kiesel und $b[d-1]$ besitzt keinen, falls er nur ein Kind hat. Eine solche teilweise Belegung mit Kieselsteinen gilt demnach auch für jeden noch zu wählenden speziellen Pfad. Sei k die Position des letzten noch mit einem Kiesel markierten Knoten von SP und j so bewählt, daß die Prozedur $Interchange$ einen zyklischen Shift der Elemente $b[1], \ldots, b[j]$ bewirkt. Demnach werden alle Kiesel auf $b[1], \ldots, b[j-1]$ entfernt.
Die Anzahl der Vergleiche, die sich durch den Aufruf von $LeafSearch$ ergeben, beträgt folglich $k+1$ und die Anzahl der Vergleiche, die $BottomUpSearch$ benötigt, ist $min\{d, d-j+1\}$, da die Wurzel nicht mit sich selber verglichen wird. Insgesamt ergeben sich also höchstens

$$k+1+d-j+1 = d+1-(d-1-k)+(d-j) \tag{15}$$

wesentliche Vergleiche.
Diese Anzahl gilt es nun wie folgt zu interpretieren:

- d ist die Länge von SP, d.h. d ist entweder genau durch die Tiefe des Heaps oder durch die Tiefe des Heaps minus Eins gegeben.
- 1 charakterisiert den Vergleich zwischen $b[0]$ und $b[d]$.
- $d-1-k$ ist die Anzahl der alten Kiesel auf dem Pfad vor dem Aufruf von $LeafSearch$ (diese Anzahl ist $d-2-k$, falls $b[d-1]$ nur ein Kind hat).
- $d-j$ ist die Anzahl von neuen Kieselsteinen auf diesem Pfad nach dem Aufruf von $LeafSearch$ (diese Anzahl ist $d-1-j$, falls $b[d-1]$ nur ein Kind hat).

Da es das Ziel ist, diese Terme als worst-case an Vergleichen aufzusummieren, gleichen sich die Terme der Sonderfälle zu den letzten zwei Punkten aus.
Der Algorithmus startet und endet mit keinem Kiesel. Wenn OP die Summe der alten Kiesel und NP entsprechend die Summe der neuen Kiesel bezeichnet, so ist NP-OP die Anzahl der verschwindenen Kiesel.

Hilfssatz 23 *Die Anzahl der verschwindenden Kiesel ist kleiner als* $\lfloor (n-1)/2 \rfloor$.

Beweis:(Wegener (1992)) Da die Kiesel nur auf den Knoten $1, \ldots, \lfloor (n-1)/2 \rfloor$ liegen können und für jeden in $MDRReHeap$ betrachteten Knoten maximal ein Kiesel verschwinden kann, folgt die Behauptung unmittelbar. □

Hilfssatz 24 *Die Anzahl der Aufrufe von BottomUpSearch ist gleich* $n + \lfloor n/2 \rfloor - 2$.

Beweis: (Wegener (1992)) Die Zahl setzt sich zusammen aus $n-1$ Aufrufen aus der Generierungsphase und $\lfloor n/2 \rfloor - 1$ Aufrufen der Auswahlphase.□

Satz 14 *Der worst-case an Schlüsselvergleichen für MDR-HEAPSORT liegt bei*

$$n \log n + (3 - c(n))n + \lfloor \log n \rfloor - 1 \leq (n+1)\log n + 1.086072n, \qquad (16)$$

mit $min\{c(n) \mid n \in N\} \approx 1.91393$.

Beweis: (Wegener (1992)) Die Ergebnisse der HEAPSORT Analyse helfen, die Summen der Tiefen der betrachteten Heaps aufzudecken:
Die Summe der Tiefen der betrachteten Heaps während der Generierungsphase ist nach Hilfssatz 18 höchstens $n-1$. Die Summe der Tiefen der betrachteten Heaps während der Generierungsphase ist nach Hilfssatz 19 höchstens $n \log n - c(n)n + \lfloor \log 2 \rfloor + 2$.
Der worst-case an Vergleichen ergibt sich aus der Summation der Terme der rechten Seite von Gleichung 15.
Die Summe der '1'-Terme entspricht der Anzahl von *BottomUpSearch*-Aufrufen und ist demnach durch Hilfssatz 24 gegeben, während die Summe über die letzten beiden Terme gerade der Anzahl der verschwindenen Kiesel gleicht, die in Hilfssatz 23 abgeschätzt wurde. Die Länge des speziellen Pfades d wird jeweils durch die Tiefe des Heaps begrenzt.
Demnach ist die Anzahl der Vergleiche höchstens:

$$\lfloor (n-1)/2 \rfloor + n + \lfloor n/2 \rfloor - 2 + n - 1 + n \log n - c(n)n + \lfloor \log 2 \rfloor + 2$$
$$\leq n \log n + (3 - c(n))n + \lfloor \log n \rfloor - 1. \square$$

6.4 Die worst-case Analyse von WEAK-HEAPSORT

Das Schlimme steht dem Besten oft zunächst. (Matthias)

Franz Grillparzer
Ein Bruderzwist in Habsburg II

Der Abschnitt über die best-case Analyse von WEAK-HEAPSORT zeigte auf, daß die im Satz 7 bewiesene untere Grenze $nk - 2^k + 1$ mit $k = \lceil \log n \rceil$ für die Anzahl der Vergleiche *scharf* ist, d.h. es gibt ein Beispiel, bei dem diese Grenze auch angenommen wird.
Gleiches gilt auch für die obere Grenze von $nk - 2^k + n - k$ Schlüsselvergleichen. Die worst-case Betrachtung kann gut auf die schon gewonnenen Resultate aus der Analyse des best-cases aufgebaut werden.
Die Idee ist, daß der erste von $MergeForest(m)$

initiierte $Merge$–Aufruf immer die Arraystellen $m-1$ und m erhält. Der spezielle Pfad verfehlt demnach die zum Fall b) führende Stelle genau um eins.
Damit läßt sich die Belegung von $Reverse[i]$ für $i \in SP$ aus der Binärcodierung von $n-2$ bestimmen.
Es lassen sich viele Sätze aus der best-case Analyse übertragen. Zum Beispiel gilt der zu Hilfssatz 5 vergleichbare

Hilfssatz 25 *Sei $n-2 = (b_k \dots b_0)_2$ mit $k = \lfloor \log(n-2) \rfloor$.
Wenn $Reverse[\lfloor \frac{n-2}{2^i} \rfloor] = b_{i-1}$ für $i \in \{1, \dots, k\}$ ist, dann ist $n-2$ das letzte Element von SP.*

Beweis: Analog zu Hilfssatz 5. □
Auf der Suche nach einem geeigneten Gegenbeispiel, das diese Bedingung nach der Generierungsphase erfüllt, findet sich die Eingabe $a[i] = i+1$ für $i \in \{0, \dots, n-2\}$ und $a[n-1] = 0$. Mit ihr läßt sich in Hinblick auf Hilfssatz 7 folgendes Resultat erzielen:

Hilfssatz 26 *Sei $n-2 = (b_k \dots b_0)_2$ mit $k = \lfloor \log(n-2) \rfloor$, $SP = \{\lfloor (n-2)/2^i \rfloor \mid i \in \{1, \dots, k\}\}$ der Weg von Index $n-2$ zu Index 1. Bei einer Initialkonfiguration $a[i] = i+1$ für $i \in \{0, \dots, n-2\}$ und $a[n-1] = 0$ gilt nach Beendigung von $WeakHeapify$:*

a) *$Reverse[0] = 0$,*

b) *$Reverse[j] = 1$, $j \notin P$,*

c) *$Reverse[n-1] = 0$, $Reverse[n-2] = 1$ und $Reverse[\lfloor \frac{n-2}{2^i} \rfloor] = b_{i-1}$, $i \in \{1, \dots, k\}$.*

Beweis: Analog zu Hilfssatz 7. □
Es tritt also bis auf einmal pro Level Fall a) ein. Auch im Fall c) markiert $n-2$ das Ende des speziellen Pfades.
Die Hilfssätze 8 und 9 gelten weiterhin, wenn $n-1$ auf $n-2$ gesetzt wird. Bei einer zusätzlich angenommenen Eingabe $a[i] = i+1, i \in \{0, \dots, n-2\}$, und $a[n-1] = 0$ gelten sogar die Hilfssätze 10 und 11, so daß sich das folgende zentrale Resultat beweisen läßt:

Folgerung 6 *Sei $n \in N$ mit $n-2 = (b_{n,k_n} \dots b_{n,0})_2$ mit $k_n = \lfloor \log(n-2) \rfloor$. Sei desweiteren $a[i] = i+1$ für $i \in \{0, \dots, n-2\}$ und $a[n-1] = 0$ eine Eingabe für WEAK-HEAPSORT. Nach dem Aufruf der Prozedur $Heapify$ ergibt sich* *im ersten $MergeForest^*$-Schritt die folgenden Kette von Transpositionen:*

$$\begin{array}{l}\tau^1_{Gparent(n-2)=0,n-2}\circ\\ \tau^{b_{n,0}\oplus b_{n-1,0}}_{Gparent(\lfloor\frac{n-2}{2^1}\rfloor)=0,\lfloor\frac{n-2}{2^1}\rfloor}\\ \circ\ldots\circ\\ \tau^{b_{n,k_n-1}\oplus b_{n-1,k_n-1}}_{Gparent(\lfloor\frac{n-2}{2^{k_n}}\rfloor)=0,\lfloor\frac{n-2}{2^{k_n}}\rfloor=1},\end{array}$$

wobei $\tau^a_{i,j}$ die a-mal angewendete Transposition von den Stellen i und j für $i,j\in\{0,\ldots,n-1\}$ bezeichnet.

Beweis: Nach Hilfssatz 25 in Verbindung mit Hilfssatz 26 tritt bei der angegebenen Sortierung im ersten $MergeForest^*$-Schritt bis auf einmal pro Level Fall a) ein. Deshalb wird als erster Schritt $\tau^1_{0,n-2}$ durchgeführt. Auch im Fall c) startet der Algorithmus mit $\tau^1_{0,n-2}$. An der Stelle $n-2$ liegt das kleinste Element (1 im Fall a) und 0 im Fall c)) von SP, so daß die Behauptung analog zu Folgerung 3 mit Hilfe der Monotonieeigenschaft erzielt werden kann. □
Zusammenfassend gilt:

Satz 15 *Sei $n\in N$ mit $n-2=(b_{n,k}\ldots b_{n,0})_2$ mit $k_n=\lfloor\log(n-2)\rfloor$. Desweiteren definiere die folgenden Permutationen:*

$$\begin{array}{l}Heapi(n):=\tau^1_{Gparent(n-2),n-2}\circ\\ \tau^1_{Gparent(n-2),n-2}\circ\tau^1_{Gparent(n-2),n-2}\circ\ldots\circ\\ \tau^{b_{n,0}}_{Gparent(\lfloor\frac{n-2}{2^1}\rfloor),\lfloor\frac{n-2}{2^1}\rfloor}\circ\tau^1_{Gparent(\lfloor\frac{n-2}{2^1}\rfloor-1),\lfloor\frac{n-2}{2^1}\rfloor-1}\circ\ldots\circ\\ \tau^{b_{n,1}}_{Gparent(\lfloor\frac{n-2}{2^2}\rfloor),\lfloor\frac{n-2}{2^2}\rfloor}\circ\tau^1_{Gparent(\lfloor\frac{n-2}{2^2}\rfloor-1),\lfloor\frac{n-2}{2^2}\rfloor-1}\circ\ldots\circ\\ \vdots\\ \tau^{b_{n,k_n-2}}_{Gparent(\lfloor\frac{n-1}{2^{k_n-1}}\rfloor),\lfloor\frac{n-1}{2^{k_n-1}}\rfloor}\circ\\ \tau^{b_{n,k_n-1}}_{Gparent(\lfloor\frac{n-1}{2^{k_n}}\rfloor)=0,\lfloor\frac{n-1}{2^{k_n}}\rfloor=1}\end{array}$$

und $Swap(0,n-1):=\tau_{0,n-1}$, letztendlich

$$\begin{array}{l}CaseAC(n):=\tau^1_{0,n-2}\circ\\ \tau^{b_{n,0}\oplus b_{n-1,0}}_{0,\lfloor\frac{n-1}{2^1}\rfloor}\circ\\ \vdots\\ \tau^{b_{n,k_n-1}\oplus b_{n-1,k_n-1}}_{0,1},\end{array}$$

wobei $\tau^a_{i,j}$ die a-mal angewendete Transposition von den Stellen i und j für $i,j\in\{0,\ldots,n-1\}$ bezeichnet.
Sei desweiteren $a[i]=i+1$ für $i\in\{0,\ldots,n-2\}$ und $a[n-1]=0$ eine Eingabe für WEAK-HEAPSORT.

Dann werden die in der Aufbauphase durchgeführten Vertauschungen korrekt durch $Heapi(n)$ und die in dem ersten Auswahlschritt durchgeführten Vertauschungen korrekt durch die Hintereinanderausführung von $Swap(0, n-1)$ und $CaseAC(n)$ beschrieben.

Beweis: Analog zu Satz 10. □
Da $Heapi(n)$ in der worst-case von der best-case Betrachtung aufgrund der unterschiedlichen Eingaben abweicht, werden diese Fälle durch die Bezeichnungen $Heapi_b(n)$ (für den best-case) und $Heapi_w(n)$ (für den worst-case) voneinander getrennt.
Die Folgerung 4 wird durch die Subsitution von $n-1$ durch $n-2$ übertragen. Nun trennen sich die Argumentationswege des best- und worst-cases langsam voneinander. Ziel ist es, das Schlüssellemma des worst-cases auf das Schlüssellemma des best-cases zu stützen. Es gilt nämlich der folgende

Hilfssatz 27

$$Heapi_w(n) = Heapi_b(n-1). \tag{17}$$

Beweis: Der Vergleich in den Sätzen 10 und 15 festgestellten Vertauschungen der Aufbauphasen liefert unmittelbar das behauptete Resultat. □

Hilfssatz 28

$$Heapi_w(n) \circ (0 \;\; n-1) = (n-2 \;\; n-1) \circ (Heapi_w(n)). \tag{18}$$

Beweis: Idee: Die Transposition $(n-2 \;\; n-1)$ soll sukzessiv durch die Kette der Transpositionen in $Heapi_w(n)$ *geschleift* werden. Die erste Transposition, die von $(n-2 \;\; n-1)$ erreicht wird, ist $(Gparent(n-2) \;\; n-2)$. Es gilt:

$$(n-2 \;\; n-1) \circ (Gparent(n-2) \;\; n-2) =$$

$$(Gparent(n-2) \;\; n-2) \circ (Gparent(n-2) \;\; n-1).$$

Außerdem beinhaltet keine der folgenden Transpositionen aus $Heapi_w(n)$ einen Teil $n-2$, so daß nun mittels Hilfssatz 27 der Beweis von Hilfssatz 14 zum gewünschten Endergebnis nachgeahmt werden kann. □

Hilfssatz 29

$$CaseAC(n) = Swap(0, n-2) \circ CaseB(n-1). \tag{19}$$

Beweis: Auch hier liefert der Vergleich in den Sätzen 10 und 15 festgestellten Vertauschungen der Aufbauphasen unmittelbar das behauptete Resultat. □

Hilfssatz 30 *(Schlüssellemma)*

$$Heapi_w(n) \circ Swap(0, n-1) \circ CaseAC(n) \circ Heapi_w(n-1)^{-1} = (n-2 \;\; n-1).$$

Beweis: Die behauptete Gleichung ist nach Hilfssatz 28 gleichbedeutend mit

$$(n-2 \;\; n-1) Heapi_w(n) \circ CaseAC(n) \circ Heapi_w(n-1)^{-1} = (n-2 \;\; n-1).$$

Die Anwendung der Hilfssätze 27 und 29 liefert die folgende äquivalente Gleichung

$$(n-2 \;\; n-1) \circ Heapi_b(n-1) \circ Swap(0, n-2) \circ CaseB(n-1) \circ Heapi_b(n-2)^{-1}$$
$$= (n-2 \;\; n-1).$$

Die Multiplikation von links mit $(n-2 \;\; n-1)$ liefert nun das Schlüssellemma des best-cases. □

Beispiel 2 *Sei* $n = 16$, *also* $n - 2 = 14 = (b_3 \;\; b_2 \;\; b_1 \;\; b_0)_2 = (1 \;\; 1 \;\; 1 \;\; 0)_2$ *und* $n - 3 = 13 = (c_3 \;\; c_2 \;\; c_1 \;\; c_0)_2 = (1 \;\; 1 \;\; 0 \;\; 1)_2$.
$Heapi_w(16) = (14 \;\; 3)^1(13 \;\; 6)^1(12 \;\; 1)^1(11 \;\; 5)^1(10 \;\; 2)^1(9 \;\; 4)^1(8 \;\; 0)^1(7 \;\; 3)^0$
$(6 \;\; 1)^1(5 \;\; 2)^1(4 \;\; 0)^1(3 \;\; 1)^1(2 \;\; 0)^1(1 \;\; 0)^1$,
$Swap(0, 16) = (0 \;\; 15)^1$,
$CaseAC(16) = (0 \;\; 14)^1(0 \;\; 7)^1(0 \;\; 3)^1(0 \;\; 1)^0$,
$Heapi(15)^{-1} = (1 \;\; 0)^1(2 \;\; 0)^1(3 \;\; 1)^0(4 \;\; 0)^1(5 \;\; 2)^1(6 \;\; 1)^1(7 \;\; 3)^1(8 \;\; 0)^1(9 \;\; 4)^1$
$(10 \;\; 2)^1(11 \;\; 5)^1(12 \;\; 1)^1(13 \;\; 6)^1$.

Damit ist

$$Heapi_w(16) \circ Swap(0, 16) \circ CaseAC(16) \circ Heapi(15)^{-1}$$
$$= (14 \;\; 15).$$

Satz 16 *Der worst-case von WEAK-HEAPSORT wird bei folgender Sortierung der Elemente erzielt:*

$$a[n-1] < a[i] < a[i+1], i \in \{0, \ldots, n-3\}.$$

Beweis: Ohne Beschränkung der Allgemeinheit sei $a[i] = i + 1$ für alle $i \in \{0, \ldots, n-2\}$ und $a[n-1] = 0$. Sei diese Eingabe als $worst(n)$–vorsortiert bezeichnet. Der Satz wird mittels vollständiger Induktion über die Anzahl der Elemente bewiesen. Induktionsanfang (n=2): Die Eingabe für WEAK-HEAPSORT ist $a[0] = 0$ und $a[1] = 1$, $r[0] = 0$ und $r[1] = 0$. Nach der Aufbauphase (es tritt genau eine Vertauschung ein) gilt: $a[0] = 1$ und $a[1] = 0$, $r[0] = 0$ und $r[1] = 1$.

Dann wird $a[2]$ auf den Wert von $a[0]$ gesetzt und das Array ist sortiert. Die Vergleichsanzahl ist 1 und beschreibt den worst-case.
Induktionsschritt$(n-1 \Longrightarrow n)$:
Das Schlüssellemma

$$Heapi_w(n) \circ Swap(0, n-1) \circ CaseAC(n) \circ Heapi_w(n-1)^{-1} = (n-2 \;\; n-1)$$

beschreibt in Verbindung mit Satz 15 den richtig beschriebenen Übergang von einem $worst(n)$–vorsortierten Array der Länge n hin zu einem $worst(n-1)$–vorsortierten Array der Länge $n-1$.
Nach Induktionsvoraussetzung tritt für ein $worst(n-1)$–vorsortiertes Array der Länge $n-1$ der worst-case ein. Es werden bis auf den Fall $n = 2^k$ genau $\lfloor \log(n+1) \rfloor$, sonst $\lfloor \log(n+1) \rfloor - 1$ Vergleiche im Auswahlschritt benötigt. Somit tritt auch für das Array der Länge n der worst-case ein. □

7 Die Anzahl von Heaps und Weak-Heaps

Wer hohe Türme bauen will, muß lange beim Fundament verweilen.

Anton Bruckner

7.1 Die Anzahl von Heaps

What is one and one and one and one and one and one and one and one and one and one?
I don't know, said Alice, I lost count.
She can't do Addition.

Lewis Caroll
Alice in Wonderland

Um in den Themenkomplex der average case Analyse einzuleiten, werden die Arbeiten von Knuth (1973) bzw. Schaffer und Sedgewick (1993) vorgestellt, die zur Bestimmung der Anzahl von Heaps führen.
Sei n die Größe des Heaps, SP der (spezielle) Pfad zum Knoten mit Index n (vergl. Abb. 10).
Die Teilbäume zu Wurzeln links bzw. rechts von SP sind vollständig.

Hilfssatz 31 *Sei $n = (b_k \ldots b_0)_2$ mit $k = \lfloor \log n \rfloor$. Die Wurzelelemente $w_i = (1b_{k-1} \ldots b_i)_2$ von SP beschreiben Teilbäume der Größe $g_i = (1b_{i-1} \ldots b_0)_2$, $i \in \{0, \ldots, k\}$.*

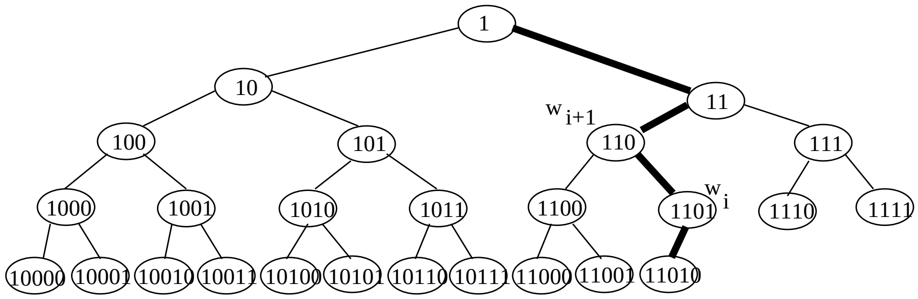

Abbildung 10: Ein Heap mit $n = 26 = (11010)_2$ Elementen.

Beweis: (Knuth (1973)[9]) Aus $k = \lfloor \log n \rfloor$ folgt direkt $b_k = 1$. Es wird obiges Resultat mittels vollständiger Induktion über i gezeigt. Für $i = 0$ ist $w_0 = (1b_{k-1} \ldots b_0)_2 = n$ und $g_0 = (1)_2 = 1$. Sei nun $w_{i+1} = (1b_{k-1} \ldots b_{i+1})_2 = \lfloor w_i/2 \rfloor$ gegeben.
Fall a) $b_i = 0$, d.h. w_i ist linkes Kind von w_{i+1}, und der rechte Teilbaum ist mit $2^i - 1$ Elementen vollständig aufgefüllt. Es gilt:

$$\begin{aligned} g_{i+1} &= g_i + 2^i - 1 + 1 \\ &= g_i + 2^i \\ &= (1b_{i-1} \ldots b_0)_2 + (10 \ldots 0)_2 \\ &= (10b_{i-1} \ldots b_0)_2 \\ &= (1b_i b_{i-1} \ldots b_0)_2. \end{aligned}$$

Fall b) $b_i = 1$, d.h. w_i ist rechtes Kind von w_{i+1} und der rechte Teilbaum ist mit $2^{i+1} - 1$ Elementen vollständig aufgefüllt. Es gilt:

$$\begin{aligned} g_{i+1} &= g_i + 2^{i+1} - 1 + 1 \\ &= g_i + 2^{i+1} \\ &= (1b_{i-1} \ldots b_0)_2 + (100 \ldots 0)_2 \\ &= (11b_{i-1} \ldots b_0)_2 \\ &= (1b_i b_{i-1} \ldots b_0)_2. \square \end{aligned}$$

Nun sollen die Anzahl und die Größen der Teilbäume rechts bzw. links von SP studiert werden.

Hilfssatz 32 *Die Anzahl der Bäume der Größe* $G_i = 2^i - 1$ *ist* $B_i = \lfloor \frac{n-2^{i-1}}{2^i} \rfloor$, $i \in \{1, \ldots, \lfloor \log n \rfloor\}$.

[9] Ebd. ist dieser Satz als Übungsaufgabe ohne Lösungshinweis gestellt.

Beweis: (Knuth (1973)[10]) Die Anzahl der Bäume gleicher Größe B_i ergibt sich durch Zählen aller Knoten mit Indizes zwischen w_i und w_{i+1}, d.h. für $i = 1$: $B_1 = \lfloor \frac{n-1}{2} \rfloor$ und für beliebiges i unter Beachtung von Satz 9:

$$\begin{aligned} B_i &= \lfloor \frac{\lfloor n/2^{i-1} \rfloor - 1}{2} \rfloor \\ &= \lfloor \frac{n/2^{i-1} - 1}{2} \rfloor \\ &= \lfloor \frac{n - 2^{i-1}}{2^i} \rfloor . \square \end{aligned}$$

Satz 17 *Sei s_i die Größe des Teilbaumes zur Wurzel i. Die Gesamtzahl der Möglichkeiten, die Zahlen $\{1, \ldots, n\}$ in einen Heap zu überführen, ist*

$$f(n) = \frac{n!}{\prod_{i=1}^{n} s_i}. \tag{20}$$

Beweis: (Schaffer und Sedgewick (1993)) Sei $f(n) = \mid \{H \mid H$ ist Heap mit n paarweise verschiedenen Schlüsseln $\} \mid$. Da die Wurzel mit n belegt werden muß und keine Restriktion für die Verteilung der übrigen Schlüssel auf die Teilbäume T_1 und T_2 wirkt, gilt folgende Rekursion:

$$f(n) = \binom{n-1}{|T_1|} f(|T_1|) f(|T_2|), \tag{21}$$

wobei $|T_1| + |T_2| = n - 1$ gilt. Durch die Division durch $n!$ ergibt sich:

$$\frac{f(n)}{n!} = \frac{1}{n} \frac{f(|T_1|)}{|T_1|!} \frac{f(|T_2|)}{|T_2|!}. \tag{22}$$

Die Expansion dieser Formel liefert:

$$f(n) = \frac{n!}{\prod_{i=1}^{n} s_i}. \square \tag{23}$$

Folgerung 7 *Sei die Eingabe von HEAPSORT durch eine Permutation der Elemente $\{1, \ldots, n\}$ gegeben. Die Ereignisse*
$E_i = \{$ Wurzel i ist maximal in dem durch ihn festgelegten Teilbaum$\}$
sind unabhängig voneinander.

[10] Ebd. ist dieser Satz als Übungsaufgabe ohne Lösungshinweis gestellt.

Beweis: Für jedes $i \in \{1, \ldots, n\}$ soll der Schlüssel zu i maximal in dem durch i festgelegten Teilbaum sein. Wenn der Baum zufällig mit den Elementen $\{1, \ldots, n\}$ gefüllt wird, ist dies nur in $1/s_i$ aller Fälle erfüllt. Da das Produkt der Randwahrscheinlichkeiten Prob(Wurzel i ist maximal in dem durch ihn festgelegten Teilbaum) gerade gleich den Bruchteil $1/(\prod_{i=1}^{n} s_i)$ der Heaps unter allen Permutationen beschreibt, sind die Ereignisse E_i unabhängig voneinander. □

Folgerung 8 *Sei $k = \lfloor \log n \rfloor$ und seien G_i, B_i bzw. $(1b_{i-1} \ldots b_0)_2$ entsprechend den Hilfssätzen definiert. Dann gilt*

$$\prod_{i=1}^{n} s_i = \prod_{i=1}^{k} (G_i)^{B_i} \cdot \prod_{i=1}^{k} (1b_{i-1} \ldots b_0)_2. \tag{24}$$

So ist z.B. die Anzahl von Möglichkeiten {A,...,Z} in Abb.10 so zu plazieren, daß die alphabetische Ordnung auf allen Pfaden gewährleistet bleibt, gleich $26!/(1^{12}3^6 7^2 15^1)(26 \cdot 10 \cdot 6 \cdot 2 \cdot 1)$

Da $s_i = 1$ für $i \in \{\lfloor n/2 \rfloor + 1, \ldots, n\}$ gilt, ist $\prod_{i=1}^{n} s_i = \prod_{i=1}^{\lfloor n/2 \rfloor} s_i$.

7.2 Die Anzahl von Weak-Heaps

Genug ist Überfluß für den Weisen.

Euripides

Ein *Allgemeiner* Heap H ist ein (nicht unbedingt binärer) Baum, mit der Eigenschaft, daß für alle $v \in H$ und dessen Kinder $w \in H$ gilt: $a[v] \geq a[w]$. Folgendes Resultat belegt, daß Weak-Heaps als Allgemeine Heaps aufgefaßt werden können:

Hilfssatz 33 *Die folgenden Bedingungen sind äquivalent:*

(W1) Der Wert an jedem Knoten ist größer oder gleich den Werten an den Knoten des rechten Teilbaumes, d.h. $\forall x \in T, \forall y \in rT(x) : a[x] \geq a[y]$.

(W1') Der Wert an jedem Knoten ist größer oder gleich den Werten seiner Großkinder, d.h. $\forall x \in T, \forall y \in S_x : a[x] \geq a[y]$.

Beweis: Die Behauptung $(W1) \Rightarrow (W1')$ ist trivial, da $(W1')$ ein Spezialfall von $(W1)$ ist.
Zur Behauptung $(W1') \Rightarrow (W1)$: Sei $x \in T$ gegeben. Annahme: Es existiert ein $y \in rT(x)$ mit $a[x] < a[y]$. Dann wähle dieses y minimal, in dem Sinne, daß folgendes gilt: Falls $a[Gparent(y)] > a[y]$ und $Gparent(y) \in rT(x)$ gilt, wird für y das Großelternteil $Gparent(y)$ gewählt. Ist dann $y \in S_x$, so gilt $a[x] < a[y]$ im Widerspruch zu (W1'). Ist $y \notin S_x$, so existiert ein $w = Gparent(y)$ mit $a[w] < a[y]$, was genauso widersprüchlich zu (W1') ist. □

Die Einbettung eines Weak-Heaps WH in einen Allgemeinen Heap H ist nun offensichtlich: Für $v \in WH$ wird ein v im Allgemeinen Heap generiert, dessen Kinder aus allen Kopien w aus WH gebildet werden, für die $w \in S_v$ gilt:

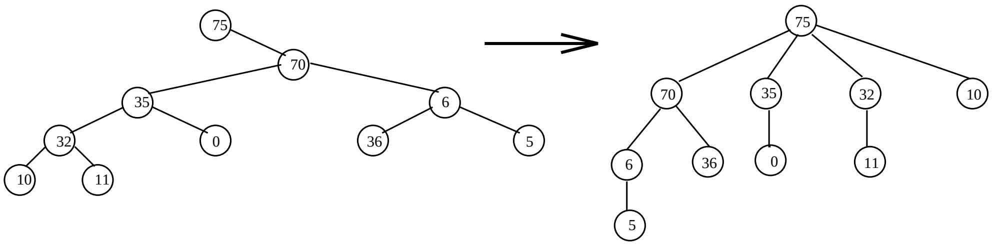

Abbildung 11: Einbettung eines Weak-Heaps in einen Allgemeinen Heap.

Satz 18 *Die Anzahl der Möglichkeiten, die Zahlen* $\{1, \ldots, n\}$ *(*$l = \lfloor \log n \rfloor$*) in einen Weak-Heap zu überführen ist im Spezialfall* $n = 2^l$

$$g(n) = \frac{n!}{2^{n-1}} \tag{25}$$

und im allgemeinen Fall

$$g(n) = \sum_{\substack{x \in \{0,1\}^{2^l} \\ |x| = n - 2^l}} \frac{(n-1)!}{\prod\limits_{i=1}^{l} \prod\limits_{j=1}^{2^{l-i}} \left(2^{i-1} + \sum\limits_{m=1}^{2^{i-1}} x_{j2^i - m} \right)}, \tag{26}$$

wobei $|x| = |(x_1, \ldots, x_n)| = x_1 + \ldots + x_n$ *ist.*

Beweis: Sei $g(N) = |\{H \mid H$ ist Allgemeiner Heap mit n paarweise verschiedenen Schlüsseln$\}|$. Da die Wurzel mit n belegt werden muß und keine Restriktion für die Verteilung der übrigen Schlüssel auf die k Teilbäume $T_1, \ldots, T_k$ wirkt, gilt folgende Rekursion:

$$g(n) = \binom{n-1}{|T_1|, \ldots, |T_k|} g(|T_1|) \cdot \ldots \cdot g(|T_k|), \tag{27}$$

wobei $|T_1| + \ldots + |T_k| = n - 1$ gilt. Division durch $n!$ ergibt:

$$\frac{g(n)}{n!} = \frac{1}{n} \frac{g(|T_1|)}{|T_1|!} \cdot \ldots \cdot \frac{g(|T_k|)}{|T_k|!}. \tag{28}$$

Die Expansion dieser Formel liefert:

$$g(n) = \frac{n!}{\prod\limits_{i=1}^{n} s_i}, \tag{29}$$

wobei s_i die Größen der durch i beschriebenen Teilbäume im Allgemeinen Heap sind. Bezeichne nun mit H den Allgemeinen Heap, der durch die Einbettung eines Weak-Heaps WHs entstand. Falls $rT(i)$ die Größe des rechten Teilbaumes in WH ist, gilt $s_i = |rT(i)| + 1$.

In dem Spezialfall $n = 2^l$ gibt es genau eine mögliche Weak-Heap Struktur, da der rechte Teilbaum der Wurzel einen vollständig binären Baum beschreibt. Zu Wurzeln der Tiefe j, $j \in \{0, \ldots, l-1\}$ existieren genau 2^{j-1} rechte Teilbäume der Größe $n/2^j - 1$.

Demnach gilt:

$$\begin{aligned}
g(n) &= \frac{n!}{n \cdot (n/2)^1 \cdot (n/4)^2 \cdot (n/8)^4 \cdot \ldots (n/2^{l-1})^{2^{l-2}}} \\
&= \frac{(n-1)! \cdot 2^{\left(\sum\limits_{i=1}^{l-1} i2^{i-1}\right)}}{(n)^{\left(\sum\limits_{i=0}^{l-2} i2^{i}\right)}} \\
&= \frac{(n-1)! \cdot 2^{\frac{1}{2}\sum_{i=1}^{l-1} i2^i}}{n^{2^{l-1}-1}} \\
&= \frac{(n-1)! \cdot 2^{\frac{1}{2}((l-2)2^l+2)}}{n^{2^{l-1}-1}} \\
&= \frac{(n-1)! \cdot 2^{(l-2)2^{l-1}+1}}{(2^l)^{2^{l-1}-1}} \\
&= \frac{(n-1)! \cdot 2^{2^{l-1}l-2^l+1}}{2^{2^{l-1}l-l}} \\
&= \frac{(n-1)! \cdot 2^{-2^l+1}}{2^{-l}} \\
&= \frac{(n-1)!}{2^{n-l-1}} \\
&= \frac{n!}{2^{n-1}}.
\end{aligned}$$

Im allgemeinen Fall gibt es $\binom{2^l}{n-2^l}$ verschiedene Weak-Heap Strukturen, da (W3) für die $n - 2^l$ Blätter der Tiefe l die genaue Positionierung nicht vorschreibt. Je nach vorgegebener Struktur variieren allerdings auch die Größen der rechten Teilbäume, die es zu majorisieren gilt. Es werden die Boole'schen Variablen $x_1, \ldots, x_{2^l}$ eingeführt, die angeben, ob ein Blatt in einer bestimmten Auswahl vorkommt oder nicht (Vergl. Abb.12).

Damit läßt sich die Summe über die $\binom{2^l}{n-2^l}$ verschiedenen zu n entsprechenden

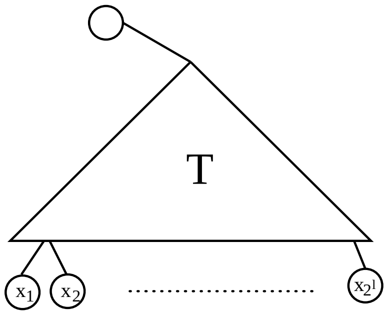

Abbildung 12: Boole'sche Variablen beschreiben die Existenz von Blättern.

Weak-Heap Strukturen wie folgt indizieren:

$$\sum_{\substack{x \in \{0,1\}^{2^l} \\ |x| = n - 2^l}} 1 \quad = \quad \binom{2^l}{n - 2^l},$$

wobei $|x| = |(x_1, \ldots, x_n)| = x_1 + \ldots + x_n$ ist.
Für ein y der Höhe h in T soll nun die Größe des rechten Teilbaumes $|rT(y)|$ berechnet werden. Bezeichne mit der Schicht i diejenigen Elemente in T, die die Höhe $h = i + 1$ besitzen, somit durchläuft i die Schichten 1 bis l. Sei y an fester Position $j \in \{1, \ldots, 2^{l-i}\}$ in der Schicht i gelegen. Durch die Einführung der Variablen $x_1, \ldots, x_{2^l}$ lassen sich damit die von y erreichbaren Blätter mit $x_{j2^i - m}$, $m \in \{0, \ldots, 2^{i-1} - 1\}$ charakterisieren. Für y gibt es insgesamt $2^{i-1} - 1$ Knoten im rechten Teilbaum, die noch ganz in T liegen. Dementsprechend gilt

$$|rT(y)| = 2^{i-1} - 1 + \sum_{m=1}^{2^{i-1}-1} x_{j2^i - m} \tag{30}$$

und damit insgesamt:

$$g(n) = \sum_{\substack{x \in \{0,1\}^{2^l} \\ |x| = n - 2^l}} \frac{(n-1)!}{\prod_{i=1}^{l} \prod_{j=1}^{2^{l-i}} \left(2^{i-1} + \sum_{m=1}^{2^{i-1}-1} x_{j2^i - m} \right)} . \Box \tag{31}$$

Folgerung 9 *Sei $\langle i, j \rangle$ der binäre Baum in Weak-Heap-Form aus Wurzel i und rechtem Teilbaum mit Wurzel j.*
Sei die Eingabe von WEAK–HEAPSORT durch eine Permutation der Elemente $\{1, \ldots, n\}$ gegeben. Die Ereignisse
$E_i = \{$ *Wurzel i ist maximal in* $\langle i, rchild(i) \rangle \}$
sind unabhängig voneinander.

Beweis: Für jedes $i \in \{1, \ldots, n\}$ soll der Schlüssel zu i maximal in $\langle i, rchild(i)\rangle$ sein. Wenn der Baum zufällig mit den Elementen $\{1, \ldots, n\}$ gefüllt wird, ist dies nur in $n!/(|rT(i)|+1)$ Fällen erfüllt. Da das Produkt der Randwahrscheinlichkeiten Prob(Wurzel i ist maximal in $\langle i, rchild(i)\rangle$) gerade gleich dem Bruchteil $1/\left(\prod_{i=1}^{n} |rT(i)|+1\right)$ der Weak-Heaps unter allen Permutationen beschreibt, sind die Ereignisse E_i unabhängig voneinander.□

Beispiel 3 *Sei* $n = 6$. *Es gibt* $\binom{4}{6-4} = 6$ *Weak-Heap Strukturen. Sind die Blättervariablen* $x_1 = 1$ *und* $x_2 = 1$ *so ergeben sich (vergl. Abb.13)* $6!/(2 \cdot 2 \cdot 6) = 30$ *Weak-Heaps, für* $x_1 = 1$ *und* $x_3 = 1$ *sind dies entsprechend* $6!/(3 \cdot 6) = 40$ *Weak-Heaps. Weiterhin ergeben sich für* $x_1 = 1$ *und* $x_4 = 1$ *20, für* $x_2 = 1$ *und* $x_3 = 1$ *20, für* $x_2 = 1$ *und* $x_3 = 1$ *10 und für* $x_2 = 1$ *und* $x_3 = 1$ *15, also insgeamt* $30 + 40 + 20 + 20 + 10 + 15 = 135$ *verschiedene Weak-Heaps.*

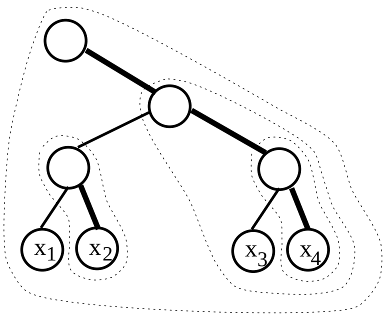

Abbildung 13: Allgemeine Weak-Heap Struktur für $n = 5, 6, 7, 8$.

Für $n = 5, 6, 7, 8$ *gilt die Formel:*

$$g(n) = \sum_{\substack{x \in \{0,1\}^4 \\ |x| = n-4}} \frac{(n-1)!}{(1+x_2)(1+x_4)(2+x_4+x_3)}. \tag{32}$$

Im Unterschied zur Betrachtung von Heaps, können nicht alle möglichen Weak-Heaps von dem Algorithmus Heapify erzeugt werden. Es gibt Binärbaumstrukturen, die (W3) erfüllen, aber nicht durch eine Vertauschung von Teilbäumen aus dem initialen von links aufgefüllten Binärbaum zu generieren sind. Diese Strukturen bzw. die darin gebetteten Weak-Heaps werden als nicht zulässig definiert.
Abb. 14 enthält ein Beispiel eines nicht zulässigen Weak–Heaps mit n = 6 Elementen.
Weder eine Drehung am Knoten a, noch am Knoten b oder c ermöglicht es, die zwei Blätter in den linken Teilbaum von a zu befördern. Demnach kann der initiale Weak-Heap, in dem eben gerade dieses gilt, nicht erreicht werden, bzw.,

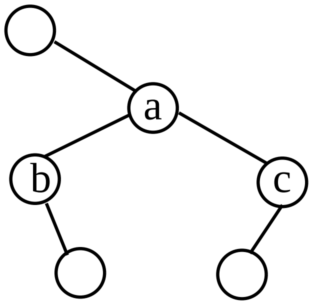

Abbildung 14: Ein nicht zulässiger Weak-Heap mit $n = 6$ Elementen.

umgedreht betrachtet, kann dieser Weak-Heap nicht aus dem ursprünglichen entstanden sein.
Die Anzahl der zulässigen Heaps mit Elementen aus $\{1, \ldots, n\}$ wird mit Hilfe einer Rückwärtsanalyse in dem nächsten Kapitel bestimmt.

8 Die Rückwärtsanalyse

Der Urgrund alles Schönen besteht in einem gewissen Einklang der Gegensätze.

Thomas von Aquin

Dieses Kapitel befaßt sich mit einer Analysemethode, die einen gegebenen Algorithmus, wie der Name schon sagt, in umgekehrter Zeitrichtung, also von der Ausgabe zur Eingabe hin, betrachtet.
Mit Hilfe der Rückwärtsanalyse läßt sich die Anzahl von möglichen Konfigurationen zu einem bestimmten Zeitpunkt im Algorithmus bestimmen und die Verteilung selbiger ermitteln.
Die bekannten Resultate über die Generierungs– und Auswahlphase (Knuth (1973) bzw. Schaffer und Sedgewick (1993)) werden in diesem Kapitel auf WEAK-HEAPSORT übertragen. Sie bilden sogar die Grundlage für die einzig geglückten Versuche von average-case Analysen der Auswahlphase von BOTTOM-UP-HEAPSORT (vergl. Kapitel 11).
Die Rückwärtsbetrachtung von Sortieralgorithmen zieht mitunter auch eine Neubewertung von Begriffen nach sich. In der von Shannon und Weaver (1949) begründeten Informationstheorie gilt, daß die Entropie, d.h. der durch Verlust an Informationen bedingte Grad der Unordnung, nicht abnehmen kann. Durch Sortieralgorithmen wird die Information über die Ordnung der Eingabe zerstört, nicht gewonnen.
Diese Erkenntnis wirft auch ein neues Licht auf ein zentrales Ergebnis der Informatik.

Bei der Bestimmung der unteren Grenze für allgemeine Sortierverfahren betrachtet man den sogenannten Entscheidungsbaum. Dabei repräsentieren die Blätter für paarweise verschiedene Schlüssel o.B.d.A. Permutationen der Elemente $\{1, \ldots, n\}$. Innere Knoten repäsentieren den zuletzt durchgeführten wesentlichen Vergleich (vergl. Wegener (1991)). Meist wird der Baum von den Blättern zur Wurzel hin betrachtet.
Die obige Erkenntnis legt jedoch nahe, den Informationsgewinn, der durch die Vergleiche entsteht, top-down zu verfolgen. Dazu seien alle Permutationen aus S_n an der Wurzel gegeben. Ein Vergleich der Stellen i und j ermöglicht die Menge S_n in zwei Mengen einzuteilen: Die Menge der Permutationen, in denen der Wert an der Stelle i größer ist als an der Stelle j und die Menge, wo dieses nicht gilt. Mehr Information liefert ein Vergleich nicht. Der Entscheidungsbaum ist somit binär und alle Permutationen müssen in ein Blatt gelangen. Damit läßt sich das bekannte Resultat über die untere Schranke unmittelbar folgern.
Genauso wird in der Berechung der Anzahl von Heaps und Weak-Heaps nach der Methode von Schaffer und Sedgewick (1993) (vergl. Satz 17 und Satz 18) die jeweilige Struktur durch die rekursiven Definition von $f(n)$ (bzw. $g(n)$) rückwärts generiert. An der Wurzel sind noch alle Festlegungen des Schlüssels für den betrachteten Knoten möglich, in den Teilbäume ist die Auswahl entspechend eingeschränkt.
Die hier vorgestellte Rückwärtsanalyse von HEAPSORT und WEAK–HEAPSORT läßt sich ähnlich zu den angeführten Beispielen veranschaulichen. Die Knoten repräsentieren weder Vergleiche noch einzelne Schlüsselfestlegungen, sondern Konfigurationen des Algorithmus. Dabei stehen wie im ersten Beispiel Permutationen der Elemente $\{1, \ldots, n\}$ an den Blättern und die sortierte Ausgabe an der Wurzel. Kanten zwischen Konfigurationen stellen mehrere zu einer Einheit zusammengefaßte Algorithmenschritte dar. Für HEAPSORT und WEAK–HEAPSORT legen die Rahmenprogramme die Größe einer solchen Einheit fest. Damit ist die Tiefe im sogenannten *Konfigurationenbaum* für alle Blätter identisch. Sind alle Eingaben gleichwahrscheinlich, so läßt sich über die Größe der Teilbäume des Konfigurationenbaumes die Wahrscheinlichkeit ermitteln, daß eine bestimmte Konfiguration zu einem festgelegten Zeitpunkt im Algorithmus erzeugt wird. Aussagen über die Verteilung von Heaps bzw. Weak-Heaps werden so möglich. Ziel der Rückwärtsbetrachtung ist es, die Struktur des Konfigurationenbaum zu erkennen, indem die Algorithmenschritte rückwärtig durchgeführt werden. Faßt man die durch eine Kante repräsentierte Einheit von Algorithmenschritten als Abbildung auf, so heißt das, daß die Beziehung zwischen diesen Einheiten und den Gegenstücken eineindeutig sein muß. Die Suche nach den Umkehrabbildungen und der Beleg der Korrektheit selbiger bilden den Kern der Rückwärtsanalyse.

8.1 Die Aufbauphase von HEAPSORT

Verlust will Vorwand. (Saladin)

Gotthold Ephraim Lessing
Nathan der Weise II, 1

Die Rückwärtsanalyse wird als Beweisverfahren in der Informatik erstmalig von Knuth (1973, Theorem H) für die Analyse von HEAPSORT verwendet:

Satz 19 *Sei s_i jeweils die Größe des durch i beschriebenen Teilbaumes. Wenn die Eingabe von HEAPSORT eine zufällige Permutation der Elemente $\{1, \ldots, n\}$ ist, dann ist jeder der $n!/(\prod_{i=1}^{\lfloor n/2 \rfloor} s_i)$ möglichen Heaps eine gleichwahrscheinliche Ausgabe der Aufbauphase.*

Beweis: (Knuth (1973)) Der Idee des Beweises beruht auf einer Rückwärtsanalyse. Sei $K^i = (a[1]_i, \ldots, a[n]_i)$ das Zwischenresultat bzw. die Konfiguration von HEAPSORT nach den Aufrufen $ReHeap(j,n)$ für $j = \lfloor n/2 \rfloor, \ldots, i$. Es wird gezeigt daß genau s_i Vorgängerkonfigurationen $K_1^{i+1}, \ldots, K_{s_i}^{i+1}$ in K_i münden. Der Fall $i = 1$ ist typisch: Sei $K = a[1]_2$ und j so gewählt, daß $K = a[j]_1$. Dann folgt $a[j]_2 = a[\lfloor j/2 \rfloor]_1, a[\lfloor j/2 \rfloor]_2 = a[\lfloor j/4 \rfloor]_1$ usw. und $a[l]_2 = a[l]_1$ für jedes l, das nicht auf dem Pfad von 1 bis j liegt. Umgekehrt betrachtet liefert die Konstruktion, $(PullUp(j))$ für jedes j als Index eines Knotens in K^1 eine Konfiguration K^2, so daß

- K^2 in K^1 mittels $ReHeap(i,n)$ überführt werden kann,
- $a[\lfloor l/2 \rfloor]_2 \geq a[l]_2$, für $2 \leq \lfloor l/2 \rfloor < l \leq n$

gilt. Deshalb sind exakt $s_1 = n$ Vorgängerkonfigurationen möglich.
Betrachtet man nun den von K^i aufgespannten Konfigurationenbaum, so folgt induktiv, daß genau $s_i \cdot \ldots \cdot s_{\lfloor n/2 \rfloor}$ Initialkonfigurationen in K^i münden. Jedes dieser Blätter des Konfigurationenbaumes entspricht einer Permutation $\sigma \in S_n$.
□

Beispiel 4 *Sei $n = 5$. Dann stellt Abb. 15 einen Anfangsausschnitt des Konfigurationenbaumes dar:*

Wegener (1995) schreibt (nach einer ähnlichen Analyse)

> Hier sollten wir einen Moment innehalten und über die technischen Schwierigkeiten die Schönheit des Prozesses der Rückwärtsanalyse genießen.

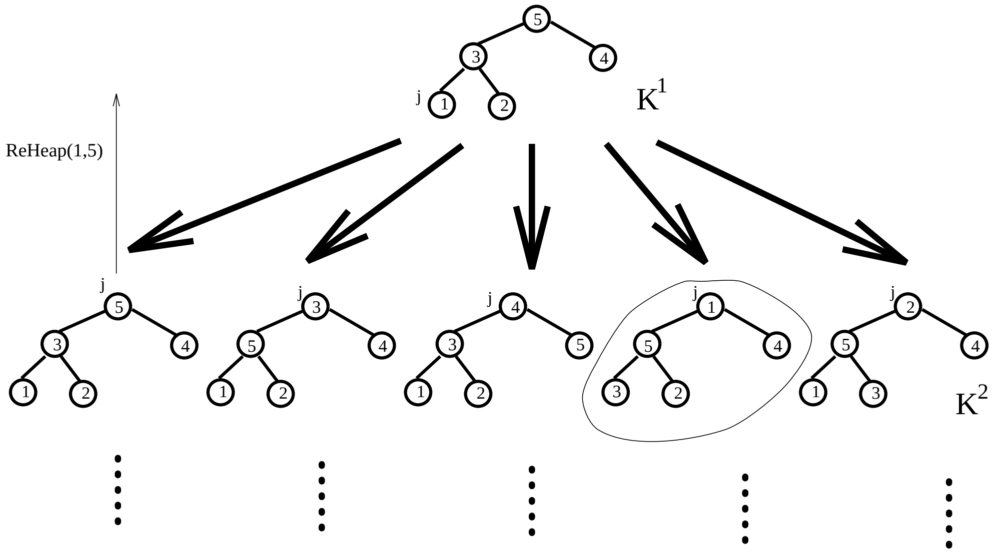

Abbildung 15: Expansionsausschnitt der Rückwärtsanalyse für die Generierungsphase von HEAPSORT.

8.2 Die Aufbauphase von WEAK-HEAPSORT

Betrachte die Dinge von einer anderen Seite, als du sie bisher sahst; denn das heißt ein neues Leben zu beginnen.

Mark Aurel
Selbstbetrachtungen, VII, 2

In Analogie zu HEAPSORT wird die Rückwärtsanalyse nun auf die Generierungsphase von WEAK-HEAPSORT übertragen. Es wird zwischen der Darstellung eines Weak-Heaps als binärer Baum und in der Arrayeinbettung unterschieden. Im ersten Fall wird Obacht darauf gelegt, daß die so erzeugten Weak-Heaps zulässig sind.

Definition 5 *Sei ein Weak-Heap WH der Größe n gegeben.* $W(n) = \{l \in WH \mid l$ *ist Wurzel von einem vollständigen Teilbaum* $\}$

Beispiel 5 *Sei $n = 7$. Die Elemente $l \in W(n)$ sind in Abbildung 16 schwarz gefärbt.*

Satz 20 *Wenn die Eingabe von WEAK-HEAPSORT eine zufällige Permutation der Elemente $\{1, \ldots, n\}$ ist, dann ist jeder mögliche und zulässige (!) Weak-Heap eine gleichwahrscheinliche Ausgabe der Aufbauphase.*
Es gibt

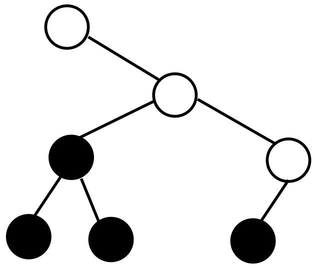

Abbildung 16: Die Wurzeln vollständiger Teilbüme.

$$A(n) = \frac{n!}{2^{|W(n)|}} \tag{33}$$

mögliche und zulässige Weak–Heaps in der Darstellung als binärer Baum.

Beweis: Die Indizes der Knoten des als binären Baum aufgefaßten Weak-Heaps seien initial o.B.d.A. mit $0, \ldots, n-1$, die Werte entsprechend mit $a[0], \ldots, a[n-1]$ bezeichnet, so daß $Merge(Gparent(i), i)$ für $i \in \{n-1, \ldots, 1\}$ gemäß Satz 2 bottom-up organisiert ist. Sei K^i die Multimenge der Knotenindizes als Zwischenresultat bzw. Konfiguration von $Heapify$ nach den Aufrufen $Merge(Gparent(j), j)$ für $j \in \{n-1, \ldots, i\}$. Die Konfiguration K^{n-1} entspricht somit einer Permutation der Elemente $\{1, \ldots, n\}$ als Eingabe des Algorithmus. Es wird s_i so bestimmt, daß s_i legale und zulässige Vorgängerkonfigurationen in K^i münden.
Abb. 17 soll die Konstruktion der zu $Merge(Gparent(i), i)$ invers stehenden Operation $MergeUp(j)$ verdeutlichen.
Es gibt zwei Fälle. Im Fall a) ist $a[Gparent(i)] > i$, so daß in der $Merge$-Prozedur kein Tausch stattfindet. Rückwärts betrachtet liefert die Wahl $j = Gparent(i)$ das an der Wurzel stehende Element der Vorgängerkonfiguration. Im Fall b) gilt $a[Gparent(i)] < i$. Es werden sowohl die Elemente $a[Gparent(i)]$ und $a[i]$ als auch die Teilbäume $rT(i)$ und $lT(i)$ getauscht. Hier liefert die Wahl $j = i$ in der Rückwärtsbetrachtung das an der Wurzel stehende Element der Ausgangskonfiguration.
Nun ist es nicht schwer, $MergeUp(j)$ anzugeben:

```
PROCEDURE MergeUp(j)
  IF j = i THEN                                   (* Fall b) *)
     Swap(a[Gparent(i)],a[i])
     Swap(lT(i),rT(i))
  FI
END MergeUp.
```

Es sei an dieser Stelle an die Prozedur $Merge(Gparent(i), i)$ erinnert, die invers zu $MergeUp(j)$ stehen soll:

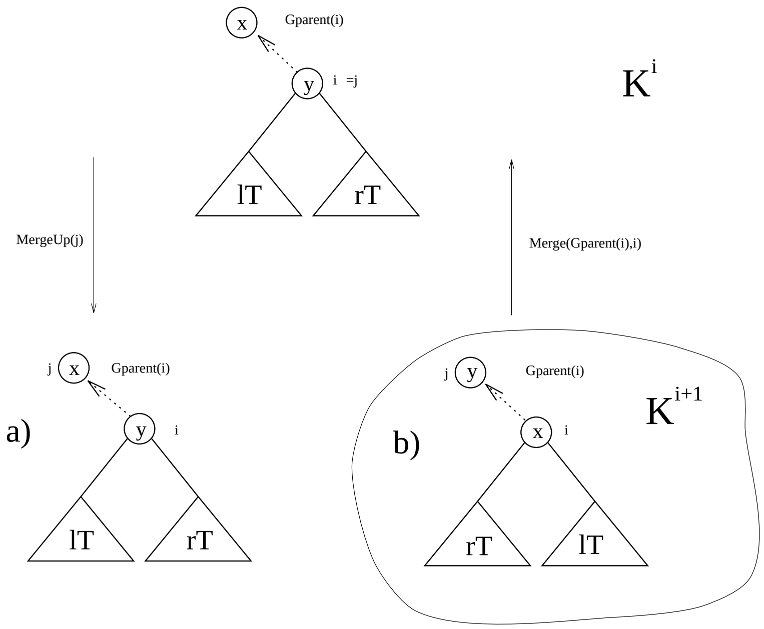

Abbildung 17: Korespondenz zwischen $Merge(Gparent(i), i)$ und $MergeUp(j)$.

```
PROCEDURE Merge(Gparent(i),i)
  IF a[Gparent(i)] < a[i] THEN                (* Fall b) *)
     Swap(a[Gparent(i)],a[i])
     Swap(lT(i),rT(i))
  FI
END Merge.
```

Wird ein $j \in \{i, Gparent(i)\}$ gewählt so ergibt der Aufruf von $MergeUp(j)$ und $Merge(Gparent(i), i)$ von einer Konfiguration K^i ausgehend über eine Konfiguration K^{i+1} letztendlich K^i zurück.
Wird aus K^{i+1} mittels $Merge(Gparent(i), i)$ eine Konfiguration K^i generiert, so wähle j so, daß der Schlüssel $a[j]$ in K^i mit $a[Gparent(i)]$ in K^{i+1} übereinstimmt. Dann liefert $MergeUp(j)$ von K^i ausgehend die ursprüngliche Konfiguration K^{i+1} zurück.
Es gilt nach dem Aufruf $MergeUp(j)$ in beiden Fällen a) und b) für alle $i + 1 \leq l \leq n - 1$: $a[Gparent(l)] \geq a[l]$, da die Bedingung in K^i für alle $i \leq l \leq n - 1$ galt.
Eine Besonderheit gilt es zu betrachten, da manche Weak-Heaps nicht zulässig sind. Nicht immer ist über beide Vorgängerkonfigurationen im Fall a) und Fall b) eine Anfangskonfiguration K^{n-1} erreichbar. Dabei wird die Anfangskonfigurationen K^{n-1} durch den von links aufgefüllten Binärbaum repräsentiert.

Es sind wieder zwei Fälle zu unterscheiden:

Fall i) i ist Wurzel eines vollständigen Teilbaumes ($i \in W(n)$).

Fall ii) i ist keine Wurzel eines vollständigen Teilbaumes ($i \notin W(n)$).

Im Fall i) existieren zu K^i beide Vorgängerkonfigurationen. Dies steht im Einklang damit, daß $Merge(Gparent(i), i)$ sowohl einen Schlüsseltausch und eine Vertauschung von Teilbäumen von i bewirkt haben kann als auch nicht, denn der zu i gehörende Teilbaum ist vollständig. Demnach verändert ein Tausch der Teilbäume nicht die Struktur des Weak-Heaps, d.h. die Struktur einer Anfangskonfiguration K^{n-1} ist in diesem Fall immer erreichbar.
Im Fall ii) existiert nur eine legale Vorgängerkonfiguration K^{i+1}, entweder die aus Fall a) oder die aus Fall b). Eine $Merge(Gparent(i), i)$-Operation kann aufgrund der Weak-Heap Struktur nur eine der folgenden Operationen bewirkt haben: Einen Schlüsseltausch und eine Vertauschung von Teilbäumen von i oder keines der beiden.
Begründung: Wurzeln von nicht vollständigen Teilbäumen treten immer nur auf einem Pfad auf. Dies gilt für die Anfangskonfiguration und wird durch $Merge(Gparent(i), i)$ nicht verletzt, da die Operation nur Teilbäume rotiert und Elemente tauscht. Sei dieser Pfad mit P bezeichnet.
Annahme: Es existiert eine Stelle i, an der durch die Rückwärtsbetrachtung von $Merge(Gparent(i), i)$ zwei oder keine Nachfolgekonfigurationen K^{i+1} beschrieben werden, die zu einem Blatt K^{n-1} führen. Wähle dieses Element minimal, d.h im geringsten Abstand zur Wurzel. Das letzte Element des Pfades P liegt nur in exakt einem Falle im gleichen Teilbaum wie durch K^{n-1} beschrieben. Keine der nachfolgenden $MergeUp(j)$-Operationen kann die Position der zu i gehörigen Teilbäume mehr verändern, Widerspruch.
Die Anzahl der erreichten zulässigen Vorgängerkonfigurationen s_i ist somit im Fall i) gleich 2 und im Fall ii) gleich 1.
Betrachtet man nun den von K^1 aufgespannten Konfigurationenbaum, so folgt induktiv, daß genau

$$\prod_{i=1}^{\lfloor n-1 \rfloor} s_i = 2^{|W(n)|}$$

Initialkonfigurationen in K^1 münden. $\square$

Beispiel 6 *Ist* $n = 5$ *liegt Fall i) in 2 Knoten vor, Fall ii) liegt entsprechend in* $n - 2 = 3$ *Knoten vor, insbesondere für den letzten* $Merge(Gparent(j), j)) = Merge(0, 1)$ *– Aufruf an Knoten mit Index 1*
Abbildung 18 veranschaulicht die Rückwärtsbetrachtung. Die umkringelten Knoten stehen für die Elemente mit vollständigen Teilbäumen. Im ersten Schritt führt die Wahl $j = 1$ *(Fall b)) in eine Sackgasse, da der erzeugte Weak-Heap*

nicht zulässig wird. Keine der Vertauschungen an den noch folgenden Knoten vermag das tief hängende Blatt nach ganz links zu befördern.
Im zweiten Schritt führt in Analogie nun die Wahl $j = Gparent(2) = 0$ zu einem nicht zulässigen Weak-Heap. Erst für die folgenden Schritte werden jeweils beide Vorgänger-Weak-Heaps zulässig.

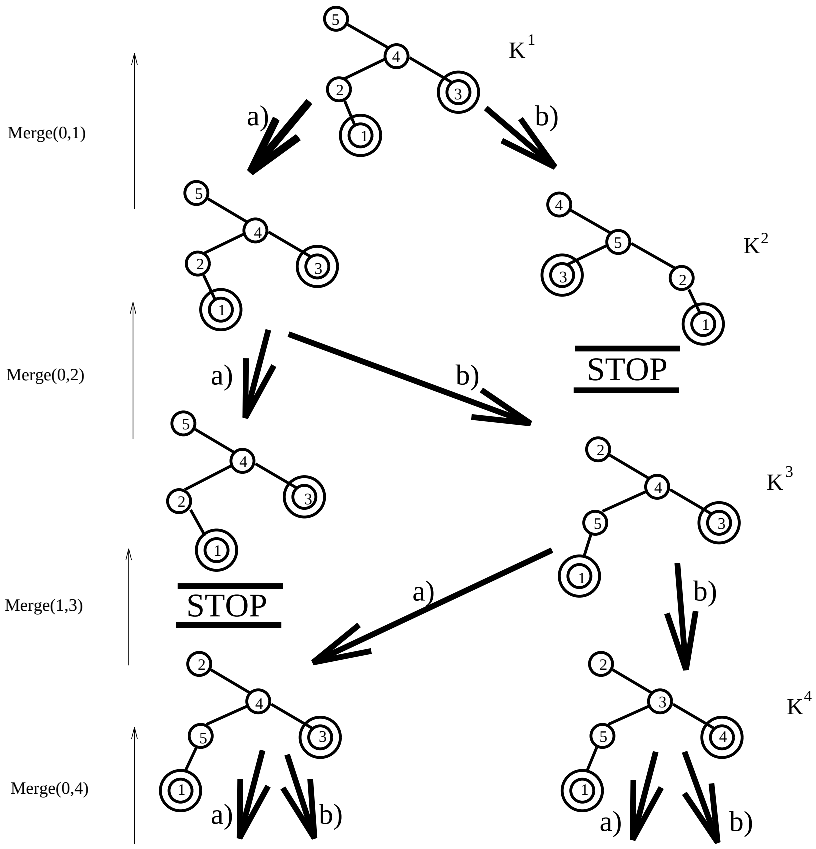

Abbildung 18: Expansionsausschnitt der Rückwärtsanalyse für die Generierungsphase von WEAK–HEAPSORT.

Die Struktur von $W(n)$ läßt sich in der Arrayeinbettung eines Weak-Heaps genau analysieren:

Satz 21 *Falls $n = 2^k$, dann beschreibt nur der Arrayindex 0 eine Wurzel von nicht vollständigen Teilbäumen. Sei $n - 1 = (b_k \dots b_0)_2 \neq 2^k - 1$ mit $k = \lceil \log(n-1) \rceil$ und $i = min\{j \mid b_j \neq 1\}$. Dann beschreiben die Indizes 0 und*

$$(b_k \dots b_j)_2, \qquad j \in \{i, \dots, k\},$$

die Wurzeln von nicht vollständigen Teilbäumen.

Beweis: Für den Index 0 existiert per Definition kein linkes Kind, demnach kann der durch 0 beschriebene Binärbaum auch nicht vollständig sein.
Für $n = 2^k$ ist der gesamte vom Index 1 beschriebener Baum und somit auch alle Teilbäume vollständig.
Sei demnach $n \neq 2^k$. Die Menge $P = \{\lfloor \frac{n-1}{2^i} \rfloor = (b_k \dots b_i)_2 \mid i \in \{1, \dots, k\}\}$ beschreibt den Weg unabhängig der durchgeführten Rotationen und Vertauschungen von dem Index $n-1$ zurück zum Index 1, da

$$\lfloor \frac{\lfloor \frac{n-1}{2^{i-1}} \rfloor}{2} \rfloor = \lfloor \frac{n-1}{2^i} \rfloor \tag{34}$$

nach Satz 9 gilt und mit $\lfloor a/2 \rfloor$ jeweils das Elternteil zu a bestimmt wird. Deshalb betrachte o.B.d.A. alle Reversebits mit 0 belegt.
Nur auf dem Pfad P treten Wurzeln von nicht vollständigen Teilbäumen auf. Solange jedoch $b_{j-1} = \lfloor \frac{n-1}{2^j} \rfloor$ MOD $2 = 1$ gilt, ist $\lfloor \frac{n-1}{2^{j-1}} \rfloor$ ein rechtes Kind von $\lfloor \frac{n-1}{2^j} \rfloor$, dessen durch ihn als Wurzel beschriebener Baum somit noch vollständig ist. Ab der Stelle $\lfloor \frac{n-1}{2^i} \rfloor = (b_k \dots b_i)_2$ mit $i = min\{j \mid b_j \neq 1\}$ ist die Vollständigkeit allerdings verletzt, da man von einem linken Kind zu $\lfloor \frac{n-1}{2^i} \rfloor$ gelangt ist. Eltern unvollständiger Bäume beschreiben unvollständige Bäume. Somit sind $(b_k \dots b_j)_2$ für alle $j \in \{i, \dots, k\}$ Wurzeln unvollständiger Bäume.□
Die Frage, ob $2^{|W(n)|}$ ein Teiler von $n!$ ist, kann nun unabhängig von der Rückwärtsanalyse bestätigt werden.

Folgerung 10 $2^{|W(n)|}$ *ist ein Teiler von* $n!$.

Beweis: Die Potenz $p_2(n)$ von 2 in der eindeutigen Primfaktorzerlegung von $n!$ läßt sich wie folgt ermitteln: Es sind $\lfloor n/2 \rfloor$ Zahlen von 1 bis n durch 2 teilbar, $\lfloor n/4 \rfloor$ durch 4, allgemein $\lfloor n/(2^k) \rfloor$ durch 2^k teilbar. Dementsprechend ist $p_2(n) = \sum_{k \geq 1} \lfloor \frac{n}{2^k} \rfloor$. Bezogen auf die Binärdarstellung von n gibt die Anzahl der Einsen an, wie oft die Zahlen $n/(2^k)$ abgerundet worden sind und demnach in der Summe an dem Werte n fehlen. Bezeichnet nun E_n die Anzahl der Einsen in der Binärdarstellung, so gilt $p_2(n) = n - E_n$.
Zu zeigen ist demnach, daß $|W(n)| \leq p_2(n)$ oder gleichbedeutend $n - |W(n)| \geq E_n$ ist. Dabei soll zwischen den Fällen $n = 2^k$ und $n \neq 2^k + 1$ unterschieden werden.
Fall 1) $n = 2^k$. Damit ist $n = (1\,0 \dots 0)_2$. Demnach ist $E_n = 1$ und $|W(n)| = n - 1$, da alle Teilbäume außer der Wurzel vollständig sind. Es gilt: $n - |W(n)| = E_n$.
Fall 2) $n \neq 2^k$. Die Binärdarstellungen von n und $n-1$ haben somit die gleiche Länge $k = \lceil (n-1) \rceil$, d.h. $n = (1\ a_{k-1} \dots a_0)_2$ und $n = (1\ b_{k-1} \dots b_0)_2$. Sei $i = min\{j \mid b_j \neq 1\}$. Demnach existieren nach Satz 21 genau $(((k-i)+1)+1) = k-i+2$ Arrayindizes, die Wurzeln von nicht vollständigen Teilbäumen sind. Also ist $n - W(n) = k-i+2$. Es ist $i = min\{j \mid b_j \neq 1\} = min\{j \neq k \mid a_j = 1\}$, da der Übertrag bei der Addition von 1 nach der Schulmethode gerade an der Stelle i

versiedet. Die Anzahl der Einsen in der Binärdarstellung von n ist kleiner-gleich $(n-i)+1$, also dem Bereich, wo noch Einsen stehen können. □
Zu den Resultaten für Weak-Heaps als binäre Bäume gesellt sich in Hinblick auf die Einbettung ein gänzlich anderes:

Satz 22 *Zwei mittels $Heapify$ aus unterschiedlichen Anfangswerten generierte Weak-Heaps können nicht sowohl in ihren Reversebits als auch in ihren Arrayinhalten übereinstimmen.*

Beweis: Die Belegung der Reversebits legt in eindeutiger Weise fest, welcher Fall in den jeweiligen $Merge$–Schritten vorlag und welche Elemente miteinander vertauscht wurden:
Ist $Reverse[i] = 0$ so folgt, daß bei $Merge(Gparent(i), i)$ die IF–Bedingung nicht erfüllt worden ist und somit wurden auch die Elemente $a[i]$ und $a[Gparent(i)]$ nicht miteinander vertauscht.
Ist hingegen $Reverse[i] = 1$, so ergibt sich eine Vertauschung der Elemente $a[i]$ und $a[Gparent(i)]$ bei $Merge(Gparent(i), i)$.
Demnach läßt sich aus jeder gegebenen Konfiguration K genau eine Anfangskonfiguration gewinnen, die mittels Heapify in K mündet: Es müssen die $Merge$–Schritte nur entsprechend zu den Reversebits rückwärts durchgeführt werden.
Algorithmisch läßt sich ein Rückwärtsschritt von $Merge(i, j)$ mittels der Prozedur $MergeUp^*(i, j)$ wie folgt realisieren:

```
PROCEDURE MergeUp*(i,j)
  IF Reverse[j] = 1 THEN
    Swap(a[i],a[j])
    Reverse[j] = 0
  FI
END MergeUp*.
```

Angenommen, es existieren zwei Weak-Heaps mit gleichen Reversebits und Arrayinhalten. Sei diejenige (eindeutig bestimmte) Kette von Vertauschungen, die sich aus der Übereinstimmung der Reversebits ergibt, durch die Permutation σ über die Indizes $0, \ldots, n-1$ bezeichnet. Bei angenommener identischer Arraybelegung nach $Heapify$, würde σ^{-1} (Permutationen sind von ihrer Natur aus bijektiv!) auf diese Arrayinhalte angewendet auch zwei identische Anfangskonfigurationen liefern, was einen Widerspruch zur Voraussetzung darstellt.□

Folgerung 11 *Es gibt genau $n!$ verschiedene eingebettete Weak-Heaps, die sich aus den $n!$ verschiedenen Permutationen gewinnen lassen. Jeder dieser Weak-Heaps ist gleichwahrscheinlich.*

Mit anderen Worten: In der Generierungsphase geht durch die Einbettung von Weak-Heaps keine Information über die Anordnung der Elemente in der Eingabe verloren.

8.3 Die Auswahlphase von HEAPSORT

Fortschritt ist nur möglich, wenn man intelligent gegen die Regeln verstößt.

Boleslaw Barlog

Die Anwendung der Rückwärtsanalyse in der Auswahlphase wurde von Sedgewick und Schaffer (1993) vorgeschlagen. Die zu $Swap(a[1], a[i+1])$ und $ReHeap(1, i)$ korespondierende inverse Transformation nennen sie $PullDown(j)$. Sie entspricht der Kombination von $PullUp(j)$ und $Swap(a[1], a[i+1])$ mit der schon in der Generierungphase angesprochenen $PullUp(j)$–Operation.

Satz 23 *Sei ein Heap $K^i = (a[1], \ldots, a[i])$ auf den i Objekten 1,...,i gegeben. Es sei s_i das Objekt an Position $\lfloor (i+1)/2 \rfloor$. Dann gibt es genau s_i Heaps $K_1^{i+1}, \ldots, K_{s_i}^{i+1}$, aus den $i+1$ Objekten 1,...,i+1, aus denen nach einer Runde der Auswahlphase K^i entsteht.*

Beweis: (Wegener (1995)[11]) Der Heap $K^i = (a[1], \ldots, a[i])$ auf den i Objekten $1, \ldots, i$ ist eine Konfiguration, die sich nach $Swap(a[1], a[l])$ und $ReHeap(1, l-1)$ für $l \in \{n, \ldots, i+1\}$ aus dem Ergebnis der Generierungsphase K^n ergibt. Es wird gezeigt, daß genau s_i Vorgängerkonfigurationen $K_1^{i+1}, \ldots, K_{s_i}^{i+1}$ in K^i münden. Zur Unterscheidung sei ein Vorgänger–Heap K^{i+1} zu K^i mit den Elementen $\overline{a}[1], \ldots, \overline{a}[i+1]$ belegt.
Es gilt $\overline{a}[i+1] \leq a[\lfloor (i+1)/2 \rfloor]$. Annahme: $\overline{a}[i+1] > a[\lfloor (i+1)/2 \rfloor]$. Nach dem Tausch $Swap(\overline{a}[1], \overline{a}[i+1])$ sinkt $\overline{a}[i+1]$ in den Heap K^{i+1} ein. Dann ist es unmöglich, daß $\overline{a}[i+1]$ das kleinere Element $a[\lfloor (i+1)/2 \rfloor]$ zum Aufsteigen zwingt, d.h. es gilt: $\overline{a}[\lfloor (i+1)/2 \rfloor] = a[\lfloor (i+1)/2 \rfloor] < \overline{a}[i+1]$.
Dies widerspricht jedoch der Heapbedingung für K^{i+1}.
Die im Satz 19 analysierte Generierungsphase von HEAPSORT belegt, daß $ReHeap(1, i)$ invers zu $PullUp(j)$ steht.
Damit steht auch die Sequenz $Swap(\overline{a}[1], \overline{a}[i+1])$ mit $ReHeap(1, i)$ invers zur Folge $PullUp(j)$ und $Swap(\overline{a}[1], \overline{a}[i+1])$. Die Prozedur $PullUp(j)$ garantiert die Heapbedingung
$\overline{a}[\lfloor l/2 \rfloor] \geq \overline{a}[l]$ für $2 \leq \lfloor l/2 \rfloor < l < i+1$
im entstehenden Heap. An die Wurzel wird das größte Element $i+1$ gesetzt und an die Stelle $i+1$ ein Element $a[j] \leq a[\lfloor (i+1)/2 \rfloor] \leq \overline{a}[\lfloor (i+1)/2 \rfloor]$. Damit gilt $\overline{a}[\lfloor l/2 \rfloor] \geq \overline{a}[l]$ für $1 \leq \lfloor l/2 \rfloor < l \leq i+1$,
d.h. K^{i+1} ist ein Heap. Somit gibt es für jedes $a[j] \leq a[\lfloor (i+1)/2 \rfloor]$ genau einen Heap K^{i+1} mit dem Objekt $a[j]$ an der Stelle $i+1$.
Deshalb gibt es genau $s_i = a[\lfloor (i+1)/2 \rfloor]$ Heaps K^{i+1}, aus denen nach einer Runde der Auswahlphase K^i entsteht. Die entstehenden Heaps sind untereinander nicht mehr gleichwahrscheinlich. □

[11] Ebd. wird das Ergebnis der Auswahlphase nicht auf das Ergebnis der Generierungsphase gestützt.

Folgerung 12 *Sei $t(j,i)$ die Größe des Teilbaumes zur Wurzel j in einem Heap der Größe i. Der Erwartungswert für $s_i = a[(i+1)/2]$ in einem Heap der Größe i ist*

$$E[s_i] = \frac{\prod_{j=1}^{\lfloor (i+1)/2 \rfloor} t(j, i+1)}{\prod_{j=1}^{\lfloor i/2 \rfloor} t(j,i)} \tag{35}$$

Beweis: Satz 19 belegt, daß es $\prod_{j=1}^{\lfloor (i+1)/2 \rfloor} t(j, i+1)$ Heaps der Größe $i+1$ und $\prod_{j=1}^{\lfloor i/2 \rfloor} t(j,i)$ Heaps der Größe i gibt. Die Division der beiden Terme ergibt somit die erwartete Anzahl von Heaps der Größe $i+1$, die mittels $Swap(1, i+1)$ und $ReHeap(1, i)$ auf einen Heap der Größe i abgebildet werden. Diese Zahl ist nach obigen Satz gleich $s_i = a[(i+1)/2]$. □

Beispiel 7 *Sei $i = 11$. Der im Satz 23 beschriebene Rückwärtsschritt $PullUp(7)$ in Verbindung mit dem anschließenden Tausch wird in Abbildung 19 dargestellt.*

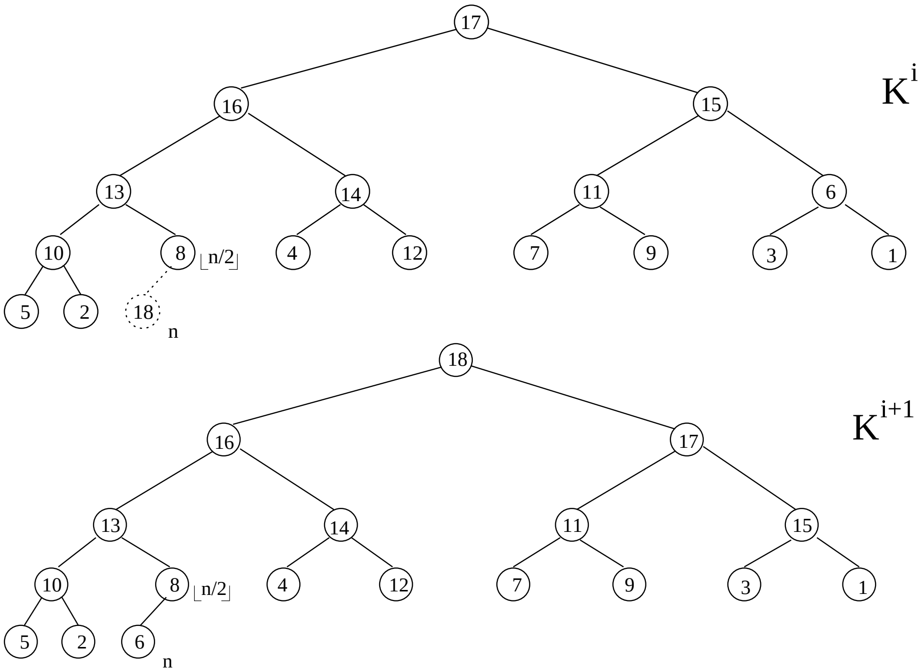

Abbildung 19: Beispiel eines rückwärtigen *ReHeap*–Schrittes.

Beispiel 8 *Sei $n = 5$. Abbildung 20 zeigt den vollständig zurückverfolgten Konfigurationenbaum der Auswahlphase.*

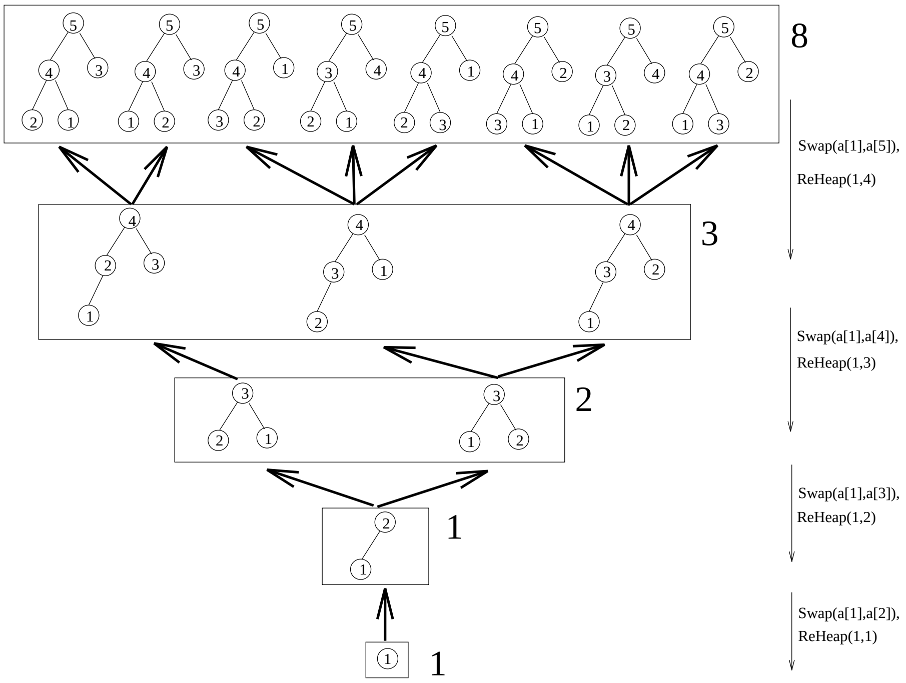

Abbildung 20: Expansions der Rückwärtsanalyse für die Auswahlphase von HEAPSORT.

8.4 Die Auswahlphase von WEAK-HEAPSORT

Wer sich am Ziel glaubt, geht zurück.

Lao-Tse
Tao-Teh-King

Die Anwendung der Rückwärtsanalyse in der Auswahlphase von WEAK-HEAPSORT läßt sich nicht direkt aus dem Ergebnis der Generierungsphase folgern, da ein $MergeForest^*$–Schritt komplexer ist als der eines $Merge(Gparent(i), i)$. Dabei sei $MergeForest^*(m)$ wie in Kapitel 3 definiert. Demnach wird eine Prozedur $MergeForestUp(j)$ als Gegenpart zu $MergeForest^*$ im Mittelpunkt der folgenden Rückwärtsanalyse stehen.

Hilfssatz 34 *Seien eine $Merge(Gparent(i), i)$–Operation und ein Index $v > i$ gegeben. Dann verändert sich $a[Gparent(v)]$ im Verlaufe der Operation nicht.*

Beweis: Sei die Menge S_i wieder wie folgt festgelegt:

$$j \in S_i :\Longleftrightarrow Gparent(i) = j.$$

Je nach Lage von v ergeben sich die folgenden Fälle:

$v \in S_{Gparent(i)}$: Da $v > i$ und $i \in S_{Gparent(i)}$ gilt, ist $v \in lT(i)$. Werden nun die Elemente $a[Gparent(i)]$ und $a[i]$ als auch die Teilbäume $lT(i)$ und $rT(i)$ miteinander getauscht, so gelangt v in die Menge S_i. Damit bleibt $a[Gparent(v)]$ unverändert. Findet kein Tausch der Elemente und Teilbäume in der $Merge(Gparent(i), i)$–Operation statt, ist diese Behauptung trivialerweise erfüllt.

$v \in S_i$: Es ist $v \in rT(i)$ und durch den möglichen Tausch der Teilbäume $rT(i)$ und $lT(i)$ gelangt v in die Menge $S_{Gparent(i)}$. Da auch die Werte $a[Gparent(i)]$ und $a[i]$ getauscht werden, bleibt $a[Gparent(v)]$ unverändert. Auch hier ist die Behauptung evident, falls das IF-Prädikat der $Merge$–Prozedur nicht erfüllt wird.

$v \notin S_i \cup S_{Gparent(i)}$: Die $Merge$–Operation verändert weder $Gparent(v)$ noch den Schlüssel, auf den $Gparent(v)$ zeigt. □

Satz 24 *Sei ein Weak-Heap $K^i = ((a[0], \ldots, a[i-1]), (r[0], \ldots, r[i-1]))$ auf den i Objekten 1,…,i gegeben. Es sei s_i das Objekt an Position $Gparent(i)$. Dann gibt es genau $2s_i$ Weak-Heaps $K_1^{i+1}, \ldots, K_{2s_i}^{i+1}$ aus den $i+1$ Objekten 1,…,i+1, aus denen nach einer Runde der Auswahlphase K^i entsteht.*

Beweis: Der Weak-Heap $K^i = ((a[0], \ldots, a[i-1]), (r[0], \ldots, r[i-1]))$ auf den i Objekten $1, \ldots, i$ ist eine Konfiguration, die sich nach $Swap(a[0], a[l])$ und $MergeForest^*(l)$ für $l \in \{n-1, \ldots, i\}$ aus dem Ergebnis der Generierungsphase K^n ergibt. Es wird gezeigt, daß genau $2s_i$ Vorgängerkonfigurationen $K_1^{i+1}, \ldots, K_{2s_i}^{i+1}$ in K^i münden. Dazu wird im folgenden $MergeForestUp(j)$ als Inverses zu $MergeForest^*$ in Idee und Realisierung beschrieben.
In rückwärtigen $Merge$–Schritten wird das Element $a[j]$ an die Wurzel bewegt und so die Menge $S_{root(T)}$ generiert (vergl. Abb. 22). Dazu startet der Algorithmus an dem Index 1 und prüft, ob j in dem linken Teilbaum liegt. Falls ja, so geht er zum linken Kind. Falls nein, so tauscht er das Element am Index 1 mit der Gesamtwurzel und ebenso die zugehörigen Teilbäume. Nun wird auch hier der Algorithmus mit dem linken Kind iterativ fortgeführt:

```
PROCEDURE MergeForestUp(j)
  x = 1
  WHILE (a[j] <> a[0]) DO
    IF (j in rT(x)) OR (a[j] = a[x]) THEN
       SWAP(a[0],a[x])
       REVERSE[x] = 1 - Reverse[x]
    FI
    x = 2x + Reverse[x]
  OD
END MergeForestUp.
```

Nach jedem Schleifendurchlauf gilt die Invarianz:

(I): Ist x der aktuell betrachtete Index, so gilt für alle $y \in lT(x)$: $a[0] > a[y]$.

Begründung: Zwei Fälle sind zu unterscheiden:

i) Ist $j \in lT(x)$, so wird ein neues $\overline{x} = 2x + Reverse[x] \in lT(x)$ gewählt. Damit bleibt für alle $y \in lT(\overline{x})$: $a[0] > a[y]$.

ii) Ist $j \in rT(x)$ so wird nach folgendem Muster getauscht:

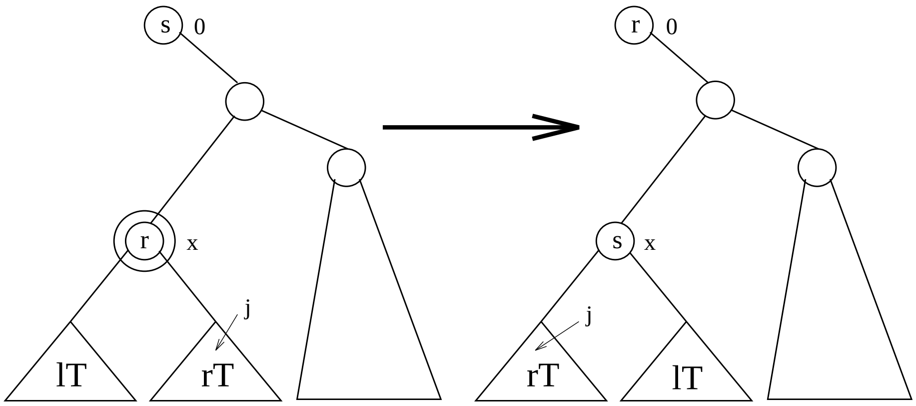

Abbildung 21: Ein rückwärtiger *Merge*–Schritt.

Damit gilt für jedes $y \in rT(x)$: $a[0] > a[y]$. Der ehemals rechte Teilbaum wird zum linken Teilbaum, d.h. für ein $\overline{x} = 2x + Reverse[x]$ und $y \in lT(\overline{x})$ gilt $a[0] > a[y]$.

Im letzten Schritt gelangt die Zahl $a[j]$ an die Wurzel. Es gilt (I), d.h. alle Schlüssel im linken Teilbaum zu j sind kleiner als der neue Wurzelwert.
Nun kann nachgewiesen werden, daß die beiden Operationen $MergeForestUp$ und $MergeForest^*$ invers zueinander wirken.
Dazu muß einerseits nachgewiesen werden, daß für ein $j \in \{0, \ldots, i-1\}$ die Aufrufe $MergeForestUp(j)$ und $MergeForest^*(i)$ von einer Konfiguration K^i ausgehend über eine Konfiguration K^{i+1} wieder K^i ergeben. Sei dazu die Konfiguration K^{i+1} durch $MergeForestUp(j)$ erreicht.
In den ersten $Merge$-Schritten von $MergeForest^*(i)$ wird aufgrund (I) $a[0]$ solange mit $a[x]$ von der letzten Stelle in $S_{root(T)}$ ausgehend verglichen, bis $x = j$ gilt. Dann werden $a[j]$ und a[0] miteinander vertauscht und das Reverse-Bit an der Stelle j gekippt.
Ist für den Index $x \leftarrow \lfloor x/2 \rfloor$ der Wert kleiner als der an der Wurzel, so erfolgt gemäß des IF-Prädikates in $Merge(0, x)$ kein Tausch der Elemente und

Teilbäume. Dies kann jedoch nur geschehen, falls rückwärtig der Fall i) vorgelegen hat, da das größere Wurzelelement nicht gegen das als kleiner bewiesene Element x ausgetauscht wurde.
Entsprechend läßt sich für den anderen Fall im *Merge*–Schritt festhalten, daß Fall ii) vorgelegen haben muß.
Die Argumentation überträgt sich mit (I) auf alle Schleifendurchläufe der Prozedur $MergeForest^*(i)$.
Wird andererseits aus K^{i+1} mittels $MergeForest^*(i)$ eine Konfiguration K^i generiert, so wähle j so, daß der Schlüssel $a[j]$ in K^i mit $a[0]$ in K^{i+1} übereinstimmt. Nun liefert $MergeForestUp(j)$ von K^i ausgehend die ursprüngliche Konfiguration K^{i+1} zurück: Liegt j im rechten Teilbaum des betrachteten Elementes x, so findet ein Tausch der Elemente x und der Teilbäume statt. Nach (I) galt vorher $a[0] > a[x]$ und somit nachher $a[x] > a[0]$, d.h. in dem dazu rückwärtig korrespondierenden $Merge(0, x)$–Schritt ist das IF-Prädikat erfüllt und so werden auch hier die Elemente und Teilbäume getauscht. Wie oben überträgt sich die Argumentation mittels (I) auf alle Schleifendurchläufe der Prozedur $MergeForestUp(j)$.
Zur Unterscheidung sei nun ein Vorgänger–Weak-Heap K^{i+1} zu K^i mit den Elementen $\overline{a}[0], \ldots, \overline{a}[i]$ belegt.
Es gilt $\overline{a}[i] \leq a[Gparent(i)]$. Annahme: $\overline{a}[i] > a[Gparent(i)]$. Nach dem Tausch $Swap(\overline{a}[0], \overline{a}[i])$ wird der Weak-Heap mittels $MergeForest^*(i)$ reorganisiert. Der Schlüssel, auf den $Gparent(i)$ zeigt, ändert sich nach Hilfssatz 34 nicht. Damit gilt $\overline{a}[Gparent(i)] = a[Gparent(i)] < a[i]$. Dies widerspricht jedoch der Weak-Heapbedingung $(W1)$ für K^{i+1}.
Die obige Analyse zeigt, daß $MergeForest^*(i)$ invers zu $MergeForestUp(j)$ steht. Damit steht auch die Sequenz $Swap(\overline{a}[0], \overline{a}[i])$ mit $MergeForest^*(i)$ invers zur Folge $MergeForestUp(j)$ und $Swap(\overline{a}[0], \overline{a}[i])$.
Die Prozedur $MergeForestUp(j)$ garantiert für alle $0 < l < i$:
$\overline{a}[Gparent(l)] \geq \overline{a}[l]$.
An die Wurzel wird das größte Element $i + 1$ gesetzt und an die Stelle i ein Element $a[j] \leq a[Gparent(i)] \leq \overline{a}[Gparent(i)]$. Damit gilt
$\overline{a}[Gparent(l)] \geq \overline{a}[l]$ für $0 \leq l \leq i$, d.h. K^{i+1} ist ein Weak–Heap. Somit gibt es für jedes $a[j] \leq a[Gparent(i)]$ genau einen Heap K^{i+1} mit dem Objekt $a[j]$ an der Stelle i.
Der jeweilige Arrayinhalt $s_i = a[Gparent(i)]$ bestimmt demnach die Anzahl der Vorgänger-Weak-Heaps. Es bleibt zu beachten, daß das *Reverse*-Bit an der Stelle i noch beliebig auf einen der Werte 0 oder 1 gesetzt werden kann. Deshalb gibt es genau $2s_i$ Heaps $K_1^{i+1}, \ldots, K_{2s_i}^{i+1}$ aus denen nach einer Runde der Auswahlphase K^i entsteht.
Der Grad der Knoten im Konfigurationenbaum kann für festes i und verschieden gewählte Konfigurationen K^i somit stark variieren. Damit sind die entstehenden Weak-Heaps untereinander nicht mehr gleichwahrscheinlich. □

Beispiel 9 *Abb. 22 veranschaulicht die Rückwärtsschritte von* $MergeForest^*$

bei dem zu generierenden Wurzelelement 1 und $n = 11$.

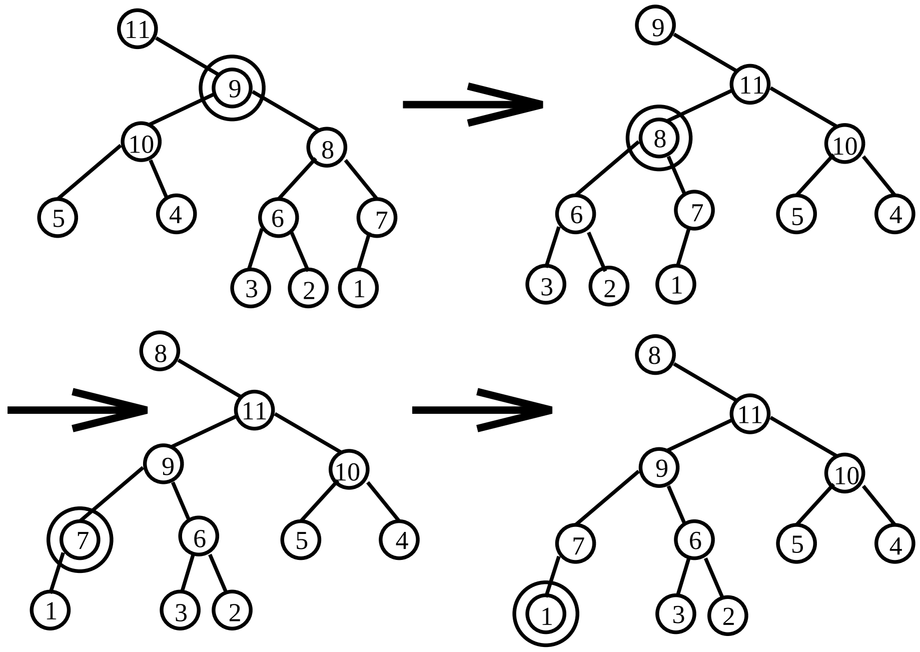

Abbildung 22: Beispiel eines rückwärtigen $MergeForest^*$–Schrittes.

Im letzen Schritt würde nur noch die 1 mit der 8 vertauscht.

Beispiel 10 *Sei* $n = 4$. *Der vollständig zurückverfolgte Konfigurationenbaum der Auswahlphase ist in Abb. 23 dargestellt: .*

9 Die Verteilung auf der Menge der Weak-Heaps

Was bleibt uns denn viel Reelles vom Leben als das Verhältnis zu vorzüglich Gleichzeitigen?

Johann Wolfgang von Goethe
an Herzog Karl August 29.6.1767

Die Anzahl, der durch einen Weak-Heap der Größe n rückwärtig zu generierenden Weak-Heaps der Größe $n+k$, $k \in \{1, \ldots, n\}$, ist nach den Betrachtungen der Rückwärtsanalyse abhängig von den Werten $a[Gparent(n)], \ldots, a[Gparent(2n-1)]$ in der entsprechenden Generierungsstufe. Die Belegung und Veränderung dieser Werte und die sich hieraus ergebenen Konsequenzen werden in diesem Kapitel untersucht.

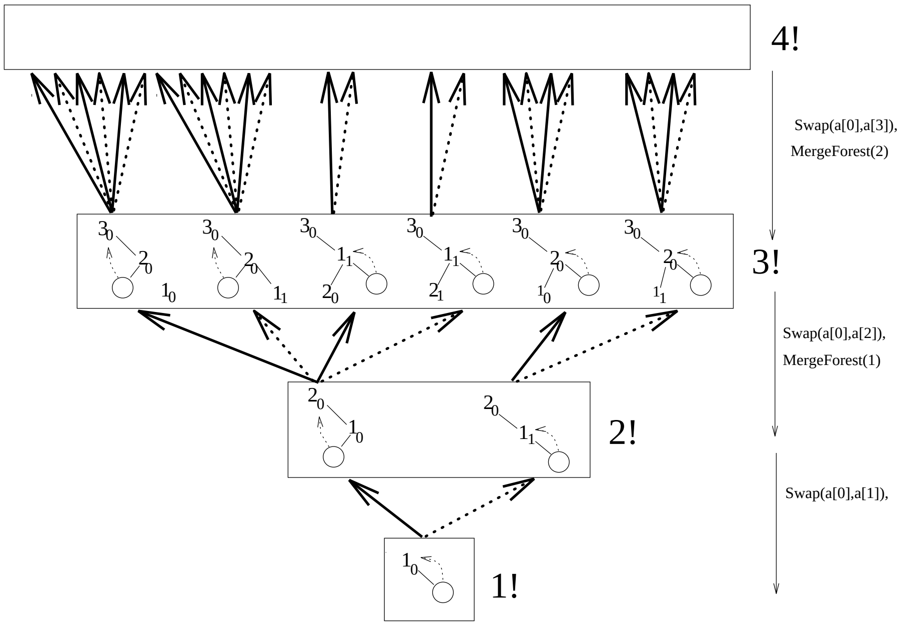

Abbildung 23: Expansions der Rückwärtsanalyse für die Auswahlphase von WEAK–HEAPSORT.

Satz 25 *Die Wahrscheinlichkeit* $Prob(a[Gparent(n)] = k)$, $k \in \{1, \ldots, n\}$, *in einem Weak-Heap auf den* n *Objekten* $\{1, \ldots, n\}$ *ist gleich* $1/n$.

Beweis: Die wesentliche Beobachtung ist die, daß der Schlüssel, auf den $Gparent(n)$ zeigt, sich in der gesamten Generierungsphase nicht ändert. Für die gesamte Aufbauphase gilt für $v = n$ Hilfssatz 34.
Zu Beginn der Generierungsphase wird die Gleichverteilung angenommen, d.h. alle Permutationen $\sigma \in S_n$ sind gleichwahrscheinlich. Dies gilt somit auch für die Belegungen von σ_i. Deshalb ist initial $P(a[i] = k) = 1/n$ für alle $i \in \{1, \ldots, n\}$, insbesondere für $i = Gparent(n)$. □
In einem Weak-Heap der Größe $2n$ sind die Indizes $Gparent(n), \ldots, Gparent(2n-1)$ kleiner als n und nur abhängig von den dort gesetzten Reversebits. Deshalb kann man selbst in einen Weak-Heap der Größe n den Indizes $Gparent(n+l)$, $l \in \{1, \ldots, n-1\}$, einen festen Wert zuordnen.

Folgerung 13 *Die Wahrscheinlichkeit* $Prob(a[Gparent(n+l)] = k)$, $l \in \{1, \ldots, n-1\}$ *und* $k \in \{1, \ldots, n\}$, *in einem Weak-Heap auf den* n *Objekten* $\{1, \ldots, n\}$ *ist gleich* $1/n$.

Beweis: Da $Parent(n+l) = \lfloor \frac{n+1}{2} \rfloor$ gilt, ist $Gparent(n+l)$, $l \in \{1, \ldots, n-1\}$, zu Beginn der Generierungsphase mit einem Schlüssel belegt.

Nun überträgt sich der obige Beweis wörtlich, indem n durch $n+l$ ersetzt wird. □

Folgerung 14 *Der Erwartungswert für $s_n = a[Gparent(n)]$ in einem Weak-Heap auf den n Objekten $\{1, \ldots, n\}$ist*

$$E[s_n] = \sum_{k=1}^{n} k\frac{1}{n} = \frac{n+1}{2}. \tag{36}$$

Ergänzend soll ein Resultat über den erwarteten Wert an einem Blatt erörtert werden.

Hilfssatz 35 *Sei j der Index eines Blattes in einem Weak-Heap auf den n Objekten $\{1, \ldots, n\}$. Nach Ablauf der Generierungsphase gilt:*

$$E[a[j]] = \frac{n+1}{3}. \tag{37}$$

Beweis: Bezeichne zur Unterscheidung mit $\overline{a}$ das Elementarray vor und mit a das Elementarray nach der Generierungsphase. Initial gilt für alle $i \in \{0, \ldots, n-1\}$: $Prob(\overline{a}[i] = k) = 1/n$. Nach Hilfssatz 6 gilt für je zwei Blätter $i \neq j$: $Gparent(i) \neq Gparent(j)$. Somit wird der Inhalt $a[j]$ für ein Blatt j durch einen einzigen Vergleich eines noch nicht veränderten Schlüssels $\overline{a}[Gparent(j)]$ mit $\overline{a}[j]$ gebildet, kurz:

$$a[j] = min\{\overline{a}[Gparent(j)], \overline{a}[j]\}. \tag{38}$$

Sei $S = \{(x,y) \in \{1, \ldots, n\}^2 \mid x \neq y\}$ die Menge von Paaren gleichwahrscheinlicher Belegungen für $\overline{a}[j]$ und $\overline{a}[Gparent(j)]$. Die Menge S ist symmetrisch in den Komponenten. Um $E[a[j]]$ auf der Menge aller Paare aus S zu berechnen, reicht es somit, $E[a[j]]$ auf der Menge $T = \{(x,y) \in \{1, \ldots, n\}^2 \mid x > y\}$ zu bestimmen. In T sind die Elemente gleichverteilt und es gilt $|T| = \frac{n(n-1)}{2}$. Demnach berechnen sich die Werte $Prob(min\{x,y\} = k)$ als Anteile der günstigen Möglichkeiten, d.h. wo $y = k$ gilt, durch Anzahl aller Möglichkeiten, d.h. durch die Größe der Menge T:

$$\begin{aligned}
Prob(min\{x,y\} = 1) &= Prob(y = 1) = \frac{2(n-1)}{n(n-1)} \\
Prob(min\{x,y\} = 2) &= Prob(y = 2) = \frac{2(n-2)}{n(n-1)} \\
\vdots \quad &\vdots \quad \vdots \\
Prob(min\{x,y\} = k) &= Prob(y = k) = \frac{2(n-k)}{n(n-1)}.
\end{aligned}$$

Damit ist

$$\begin{aligned}
E[a[j]] &= \sum_{k=1}^{n-1} k \cdot Prob(min\{x,y\} = k) \\
&= 2\sum_{k=1}^{n-1} k \frac{n-k}{n(n-1)} \\
&= \frac{2}{n(n-1)} \sum_{k=1}^{n-1} \left(kn - k^2\right) \\
&= \frac{2}{n(n-1)} \frac{n^2(n-1)}{2} - \frac{1}{3} n(n-1)(2n-1) \\
&= n - \frac{1}{3}(2n-1) \\
&= \frac{1}{3}(n+1). \square
\end{aligned}$$

Satz 26 *Sei ein Weak-Heap auf den i Objekten $\{1,\ldots,i\}$ gegeben. Die Wahrscheinlichkeit, daß in einer Runde der Rückwartsanalyse für die Auswahlphase $2j$, $j \in \{1,\ldots,i\}$, Weak-Heaps auf den $i+1$ Objekten $\{1,\ldots,i+1\}$ generiert werden, ist gleich $1/i$.*

Beweis: Nach Folgerung 11 gibt es $i!$ Weak-Heaps auf den i Objekten $\{1,\ldots,i\}$ und entspechend $(i+1)!$ Weak-Heaps auf den $i+1$ Objekten $\{1,\ldots,i+1\}$. Nach Satz 24 landen $2 \cdot a[Gparent(i)]$ Weak-Heaps der Größe $i+1$ auf einem Weak-Heap der Größe i. Nach Satz 25 ist die Verteilung der $a[Gparent(i)] \in 1,\ldots,i$ gleich. Somit landen auf

$$\begin{aligned}
\frac{(i+1)!}{\sum_{j=1}^{i} 2j} &= \frac{(i+1)!}{2\sum_{j=1}^{i} j} \\
&= \frac{(i+1)!}{i(i+1)} \\
&= (i-1)!
\end{aligned}$$

Weak-Heaps der Größe i nach einer Runde der Auswahlphase gleichwahrscheinlich 2,4, bis hin zu $2i$ Weak-Heaps der Größe $i+1$. Damit ist die Wahrscheinlichkeit, daß in einer Runde der Rückwartsanalyse für die Auswahlphase $2j$, $j \in \{1,\ldots,i\}$, Weak-Heaps der Größe $i+1$ generiert werden, gleich $1/i$. $\square$

Satz 27 *Sei $W(n,s)$ die Menge aller Weak-Heaps auf den n Objekten $\{1,\ldots,n\}$ mit*
$a[Gparent(n)] = s$. Dann gibt es

$$4(n-2)!\left((n+1)(s-1) + \sum_{t=1}^{s-1} t(s-1) + \sum_{t=s+1}^{n} ts\right)$$

Weak-Heaps auf den $n+2$ Objekten $\{1,\ldots,n+2\}$, die von $W(n,s)$ ausgehend in zwei Schritten der Rückwärtsanalyse generiert werden.

Beweis: Da nach Satz 24 im zweiten Schritt der Rückwärtsanalyse $2\cdot a[Gparent(n+1)]$ Weak-Heaps erzeugt werden, gilt es, die Veränderung dieses Wertes schon im ersten Schritt zu verfolgen.
In der $MergeForest^*$– und somit auch in der $MergeForestUp$–Prozedur verändern die Bewegungen eines ausgesuchten Wertes zur Wurzel hin nach Hilfssatz 34 nicht die Schlüssel, auf die $Gparent(n)$ und $Gparent(n+1)$ zeigen.
Ein Schritt der Rückwärtsanayse beinhaltet einen Aufruf von $MergeForestUp$ und eine abschließende Vertauschung des Wurzelwertes mit dem auf $n+1$ festgelegten Wert der Stelle n, kurz: $Swap(a[0], a[n])$. Falls zu diesem Zeitpunkt $Gparent(n+1)$ auf den Index 0 zeigt, so wird dieser Tausch den Schlüssel $a[Gparent(n+1)]$ verändern.
Da $Gparent(n+1) \neq Gparent(n)$ und $Gparent(n+1) \leq Parent(n+1) < n$ ist, gilt auch nach dem ersten Schritt der Rückwärtsanalyse $a[Gparent(n+1)] \neq s$. Demnach werden im zweiten Schritt von keinem Weak-Heap der Größe n ausgehend genau s Weak-Heaps der Größe $n+2$ generiert. Demnach können die Belegungen von $t = a[Gparent(n+1)]$ in die Fälle $t > s$ und $t < s$ unterschieden werden.

$t > s$: $MergeForestUp(Gparent(n+1))$ kann im ersten Schritt nicht aufgerufen worden sein, da dies zu einem nicht zulässigen Weak-Heap führen würde. Demnach wird $Gparent(n+1)$ nie 0 und es gibt $2t$ Weak-Heaps, die im zweiten Schritt generiert werden.

$t < s$: Es werden im zweiten Schritt $2t$ Weak-Heaps generiert, falls nach dem ersten Schritt $Gparent(n+1) \neq 0$ gilt. In einem der s möglichen Belegungen für t ist $Gparent(n+1) = 0$, d.h. es wird $a[Gparent(n+1)]$ durch $Swap(0,n)$ auf $n+1$ gesetzt. Somit werden im zweiten Schritt $2(n+1)$ Weak-Heaps generiert.

Nach Satz 25 und Folgerung 13 ist

$$Prob(a[Gparent(n)] = k) = Prob(a[Gparent(n+1)] = k) = 1/n$$

unabhängig vom gewählten $k \in \{0,\ldots,n\}$. Dies gilt somit auch nach dem ersten $MergeForestUp$–Schritt. Dementsprechend liegen die Fälle $t = k$, $k \in \{1,\ldots,n\} - \{s\}$, für die Menge $W(n,s)$ gleich häufig vor, nämlich

$$\frac{2s\mid W(n,s)\mid}{n-1} = \frac{2s(n-1)!}{n-1} = 2s(n-2)!$$

mal.
Unter der Bedingung $t < s$ lassen sich aus der Menge $W(n,s)$ insgesamt

$$2s(n-2)!\sum_{t=s+1}^{n} 2t$$

Weak-Heaps der Größe $n+2$ erzeugen
Unter der Bedingung $t > s$ tritt der Fall $Gparent(n{+}1) = 0$ nach $Swap(0,n)$ mit einer Wahrscheinlichkeit von $1/s$ und der Fall $Gparent(n{+}1) \neq 0$ entsprechend mit einer Wahrscheinlichkeit $(s-1)/s$. in diesem Fall lassen sich aus der Menge $W(n,s)$ also insgesamt

$$2s(n-2)! \sum_{t=s+1}^{n} \left(\frac{s-1}{s} 2t + \frac{1}{s} 2(n+1) \right)$$

Weak-Heaps der Größe $n+2$ erzeugen
Zusammmengenommen sind dies

$$2(n-2)! \left(\sum_{t=1}^{s-1} ((s-1)2t + 2(n+1)) + \sum_{t=s+1}^{n} s \cdot 2t \right)$$

viele. □
Es werden rückwärtig somit

$$\sum_{s=1}^{n} 4(n-2)! \left((n+1)(s-1) + \sum_{t=1}^{s-1} t(s-1) + \sum_{t=s+1}^{n} ts \right)$$

$$\begin{aligned}
&= 4(n-2)! \left((n+1) \sum_{s=1}^{n} (s-1) + \sum_{s=1}^{n} \sum_{t=1}^{s-1} t(s-1) + \sum_{s=1}^{n} \sum_{t=s+1}^{n} ts \right) \\
&= 4(n-2)! \left((n+1) \sum_{s=0}^{n-1} s + \sum_{t=1}^{n} t \sum_{s=t+1}^{n} (s-1) + \sum_{t=1}^{n} t \sum_{s=1}^{t-1} s \right) \\
&= 4(n-2)! \left((n+1) \binom{n}{2} + \sum_{t=1}^{n} t \left(\sum_{s=t+1}^{n} (s-1) + \sum_{s=1}^{t-1} s \right) \right) \\
&= 4(n-2)! \left((n+1) \binom{n}{2} + \sum_{t=1}^{n} t \sum_{s=1}^{n-1} s \right) \\
&= 4(n-2)! \left((n+1) \binom{n}{2} + \sum_{t=1}^{n} t \binom{n-1}{2} \right) \\
&= 4(n-2)! \left(\binom{n}{2} \sum_{t=1}^{n+1} t \right) \\
&= 4(n-2)! \left(\binom{n}{2} \binom{n+2}{2} \right) \\
&= (n+2)!
\end{aligned}$$

Weak-Heaps der Stufe $n+2$ generiert.

Die Aussage des Satzes erlaubt eine Einsicht in die Größe des Konfigurationenbaumes, der in zwei Schritten der Rückwärtsanalyse für eine Menge von Weak-Heaps mit einer bestimmten Eigenschaft generiert wird.
Die genaue Verteilung der Weak-Heaps der Größe n ergibt sich unmittelbar aus der Größe der durch die einzelnen Elemente aufgespannten Konfigurationenteilbäume. Diese zu ermitteln, erweist sich als nicht einfach, da sich die Sonderfälle, bei denen ein $Gparent(n+l)$ auf die Wurzel zeigt, überlagern und somit für einen exponentiellen Blow-up an Fallunterscheidungen führen:

Beispiel 11 *Sei der Ausgangsgrad eines Knotens in dem durch die Rückwärtsanalyse aufgespannten Konfigurationenbaum gleich der Anzahl der in einem Schritt von dem Knoten aus erzeugbaren Weak-Heaps.*
Es soll für einen Weak-Heap der Größe n mit den Werten $a[Gparent(n)] = s$, $a[Gparent(n+1)] = t$ *und* $a[Gparent(n+2)] = u$ *die Anzahl, der in der Rückwärtsanalyse in drei Schritten zu generierenden Weak-Heaps ermittelt werden. Sei dazu* $Position(k)$ *eine Funktion, die zu einem vorgegebenen Wert den aktuellen Index bestimmt.*
Dabei sollen die Prozeduraufrufe $MergeForestUp(Position(k_1))$, $Swap(0,n)$, $MergeForestUp(Position(k_2))$ *und* $Swap(0,n+1)$ *für alle möglichen Paare* (k_1,k_2) *hintereinander initiiert werden. Die zu entsprechenden Zeitpunkten vorliegenden Werte an den Stellen* $Gparent(n)$ *(initial),* $Gparent(n+2)$ *(nach dem* $MergeForestUp(Position(k_1))$*– und* $Swap(0,n)$*–Schritt) und* $Gparent(n+2)$ *(nach dem* $MergeForestUp((Position(k_2))$*– und* $Swap(0,n+1)$*–Schritt) ergeben durch jeweilige Verdoppelung und Multiplikation untereinander die Teilbaumgröße für das gewählte Paar* (k_1,k_2)*. Betrachte folgende Fallunterscheidung*
Fall 1) $k_1 = t$*, d.h.* $Gparent(n+1)$ *zeigt nach* $MergeForestUp(Position(k_1))$ *auf die Wurzel. Demnach ist* $a[Gparent(n+1)]$ *nach dem ersten Schritt der Rückwärtsanalyse gleich n+1. Ist* $k_2 = u$ *so ergibt sich nach dem zweiten Schritt ein Ausgangsgrad von* $2(n+2)$*. Diese Kombination tritt* $2\cdot 2$ *mal auf. Ist hingegen* $k_2 \neq u$ *so ergibt sich ein Ausgangsgrad von* $2u$*. Letztere Situation tritt* $2n$ *mal auf.*
Fall 2) $k_1 = u$*, d.h.* $Gparent(n+2)$ *zeigt nach*
$MergeForestUp(Position(k_1))$ *auf die Wurzel. In allen* $2t$ *auftretenden Fällen des ersten Schrittes ist der sich ergebene Ausgangsgrad des zweiten Schrittes gleich* $2(n+1)$*.*
Fall 3) $k_1 \notin \{t,u\}$*. Hier muß wieder zwischen dem Situationen* $k_2 = u$ *und* $k_2 \neq u$ *unterschieden werden. Im ersten* $2(s-2)\cdot 2$ *mal vorliegenden Fall ist der gesuchte Grad* $2(n+2)$ *und im zweiten* $2(s-2)\cdot 2(t-1)$ *mal vorliegenden Fall ergibt sich hingegen* $2u$*.*
Letztendlich müssen die Werte der einzelnen Fälle noch aufaddiert werden, um die Größe des durch den Weak-Heap in drei Schritten erzeugten Teilbaum zu gewinnen.

Beschränkt man sich auf eine Negativaussage, so gilt folgendes Resultat:

Satz 28 *Sei ein Weak-Heap* WH *auf den* n *Objekten* $\{1, \ldots, n\}$ *gegeben. Von* WH *ausgehend tritt in der Rückwärtsanalyse auf Wegen der Länge* n *kein Ausgangsgrad doppelt auf.*

Beweis: Bezeichne mit s_l, $l \in \{1, \ldots, n\}$, den Wert $a[Gparent(n + l)]$. Dabei beschreibt s_l den Ausgangsgrad in der jeweiligen Generationsstufe. Nach Hilfssatz 6 gilt, daß die Indizes $Gparent(n + l)$ in WH unabhängig von dem Algorithmenstadium paarweise verschieden sind.
In der $MergeForestUp$–Prozedur verändert die Bewegung eines ausgesuchten Wertes zur Wurzel hin nicht die Schlüssel, auf die $Gparent(n+l)$, $l \in \{1, \ldots, n\}$, zeigen.
Der k-te Schritt der Rückwärtsanalyse, $k \in \{1, \ldots, n\}$, beinhaltet einen Aufruf von $MergeForestUp$ und eine abschließende Vertauschung des Wurzelwertes mit dem auf $n+k$ festgelegten Wert der Stelle $n+k-1$. Falls zu diesem Zeitpunkt $Gparent(n+l)$ für ein $l \in \{1, \ldots, n\}$ auf den Index 0 zeigt, so wird dieser Tausch den Schlüssel s_l auf $n+k$ setzen. Dabei ist für alle $j \in \{1, \ldots, n\} - \{l\}$: $n+k > s_j$
In der l-ten Stufe der Rückwärtsanalyse ist der Wert s_l somit entweder unverändert oder auf einen Wert $n + k$, $k \in \{1, \ldots, n\}$ gesetzt. Nie jedoch entspricht er einem Wert s_j für $j \in \{1, \ldots, n\} - \{l\}$. Die Werte s_l, $l \in \{1, \ldots, n\}$, in WH bleiben also auf allen Wegen der Länge n der Rückwärtsanalyse paarweise verschieden, kein Ausgangsgrad tritt also doppelt auf. □
Die Ausgangsgrade im Konfigurationenbaum unterscheiden sich in einem festgehaltenen Level stark voneinander doch belegt Satz 28, daß dieses Mißverhältnis in mehreren $MergeForest$– Schritten zum Teil wieder kompensiert wird.
Zusammenfassend kann demnach festgehalten werden, daß die Auswahlphase von WEAK-HEAPSORT dazu neigt, die nach der Aufbauphase bestehende Gleichverteilung von Weak-Heap in mehreren Schritten wieder anzunähern.

10 Die average-case Analysen der Aufbauphase

Nichts ist beständig! Manches Mißverhältnis löst unbemerkt, indem die Tage rollen, durch Stufenschritte sich in Harmonie.

Johann Wolfgang von Goethe
Die natürliche Tochter IV, 2

10.1 Aufbauphase von HEAPSORT

Man soll die Dinge nicht so tragisch nehmen, wie sie sind.

Karl Valentin

Der Aufbau eines Heaps erhält die Gleichverteilung, d.h. jeder der entstehenden Heaps birgt die gleiche Wahrscheinlichkeit, generiert zu werden. Dadurch läßt sich die Durchschnittszahl an Vergleichen ermitteln, die in der Aufbauphase notwendig ist.
Doberkat (1980 und 1984) wählt hierzu einen auf den ersten Blick befremdenden Ansatz zur Analyse, den er selbst wie folgt charakterisiert:

> The algorithm is regarded as a kind of measure transfoming machine, which has as input the distribution of the random variables, on the realizations of which the algorithm is to be performed, and as output the image of the measure with respect to the function which is computed by the algorithm.

Es wird angenommen, daß die n Eingaben des Algorithmus aus einer symmetrischen Menge A stammen, die die folgenden Eigenschaften hat:

1. $A \subset R^n$ ist eine Borel Menge, d.h. ein Element der kleinsten σ-Algebra $\mathcal{B}$ über R^n, die jede offene Menge noch ganz enthält.
2. Wenn $x \in A$ ist, dann gilt für $i \neq j$: $x_i \neq x_j$.
3. Wenn $x = (x_1, \ldots, x_n)^T \in A$ ist, dann ist für alle $\sigma \in S_n$ auch $(x_{\sigma(1)}, \ldots, x_{\sigma(n)})^T \in A$.

Bedingung 1) ermöglicht es, ein Wahrscheinlichkeitsmaß auf A zu definieren. Bedingung 2) gilt bei einer kontinuierlichen Wahrscheinlichkeitsverteilung fast–überall (d.h. bis auf eine Menge mit dem Maß Null). Bedingung 3) sichert, daß die Ausgabe des Algorithmus nicht aus A 'entwischt'.
Als Hilfsmittel werden zwei zentrale Sätze aus der Analysis im R^n benötigt: Der Transformations– und der Eindeutigkeitssatz.

Satz 29 *Sei X offene Menge im R^n und $f : X \to Y$ ein C^1-Diffeomorphismus (bijektive, stetig differenzierbare Abbildung mit $detF'(x) \neq 0$). Dann gilt:*

- *f ist (Lebesque–)integrierbar über Y genau dann, wenn $(f \circ F) \mid detF' \mid$ (Lebesque–)integrierbar über X ist und*

- *$\int_Y f(y)dy = \int_X (f \circ F(x)) \mid detF'(x) \mid dx$.*

Beweis: (Rudin (1974)).□

Satz 30 *Seien μ^1, μ^2 Wahrscheinlichkeitsmaße auf der von A induzierten Borel Menge $\mathcal{B}$, so daß für alle stetigen und beschränkten Abbildungen $g : A \to R$ gilt:*

$$\int_A g d\mu^{(1)} = \int_A g d\mu^{(2)}.$$

Dann ist $\mu^{(1)} = \mu^{(2)}$ für alle $A \in \mathcal{B}$.

Beweis: (Rudin (1974)).□

Es wird angenommen, daß die Verteilung μ der Elemente in A die Dichte f besitzt, d.h. für alle Borel Mengen $B \subset A$ gilt:

$$\mu(B) = \int_B f(x)dx. \tag{39}$$

Das Paar (A, μ) wird als symmetrisches Modell (Doberkat (1983)) bezeichet. Ein Vektor $x \in A$ heißt k-Heap, wenn der durch k beschriebene Teilbaum die Heapeigenschaft erfüllt. Desweiteren bezeichne mit A_k die Menge aller k-Heaps. Wenn nun auf $y \in A_{k+1}$ $ReHeap(k)$ angewendet wird, so ergibt sich ein k-Heap, an dessen Wurzel ein Element j steht. Die Menge $A_{k,j}$ bestehe aus den $(k+1)$-Heaps, die diese Eigenschaft haben. Dann ergibt sich das folgende Resultat in Analogie zu Satz 19:

Hilfssatz 36 *Sei $R_k : A_{k+1} \to A_k$, die zu $ReHeap(k)$ korrespondierende Abbildung, dann ist $R_k : A_{k,j} \to A_k$ für alle j im Teilbaum zur Wurzel k eine Bijektion.*

Beweis: (Doberkat (1984)) R_k beschreibt den Algorithmus im k–ten Schritt der Aufbauphase. Bezeichne mit s_k die Anzahl der Elemente in dem durch k beschriebenen Teilbaum. Im aufgespannten Konfigurationenenbaum wurde gezeigt daß genau s_k Vorgängerkonfigurationen $K^1_{k+1}, \ldots, K^{s_k}_{k+1}$ in K_k münden, bzw. die Zuordnung Konfigurationen–Vorgängerkonfiguration bei festgehaltenen $j \in \{1, \ldots, s_k\}$ eineindeutig war. □

Demnach hat jeder k-Heap genau $C_k = s_{\lfloor n/2 \rfloor + 1} \cdot \ldots \cdot s_k$ Urbilder unter R_k.

Sei $\mu^{(k)}$ die Wahrscheinlichkeitsverteilung aller k-Heaps, d.h.

$$\begin{aligned} \mu^{(\lfloor n/2 \rfloor + 1)} &= \mu \\ \mu^{(k)} &= R_k(\mu^{(k+1)}). \end{aligned}$$

Der durch k beschriebene Teilbaum sei im folgenden mit T_k bezeichnet.

Satz 31 *Der Algorithmus erhält die Orginalverteilung bis auf einen konstanten Faktor:*

$$\begin{aligned} \mu^{(k)}(A) &= \left(\prod_{i=k}^{\lfloor n/2 \rfloor} s_j \right) \cdot \mu(A) \\ &= C_k \cdot \int_A f(x)dx. \end{aligned}$$

Beweis: (Doberkat (1984)) Dies ist eine Folgerung aus Satz 19, da für ein symmetrisches Modell (vergl. Doberkat (1993)) die gleiche Verteilung bezüglich einer vordefinierten Ordnung wie das diskrete Modell liefert, sofern der Algorithmus die Eingabe nur entsprechend dieser Ordnung verändert.
Doberkat gibt auch einen von Satz 19 unabhängigen Beweis, den er durch Rückwärtsinduktion über k startend bei $k = \lfloor n/2 \rfloor + 1$ führt. Da $s_{\lfloor n/2 \rfloor+1} = 1$ gilt für den Induktionsanfang:

$$\mu^{(\lfloor n/2 \rfloor+1)}(A) = P(\sigma \in A) = \int_A f(x)dx.$$

Sei die Aussage für $k+1 \leq \lfloor n/2 \rfloor + 1$ bewiesen und sei $h : A_k \to R$ eine stetige und beschränkte Abbildung. Dann gilt:

$$\begin{aligned}
\int_{A_k} hd\mu^{(k)} &\overset{Urbilder}{=} \int_{R_k^{-1}(A_k)} h \circ R_k d\mu^{(k+1)} \\
&\overset{Hypoth.}{=} C_{k+1} \int_{A_{k+1}} h(R_k(x))f(x)dx \\
&\overset{fsymm.}{=} C_{k+1} \int_{A_{k+1}} h(R_k(x))f(R_k(x))dx \\
&\overset{Partit.von A_{k+1}}{=} C_{k+1} \sum_{j \in T_k} \int_{A_{k+1,j}} h(R_k(x))f(R_k(x))dx \\
&\overset{HS36, Satz29}{=} C_{k+1} \sum_{j \in T_k} \int_{A_k} h(x)f(x)dx \\
&\overset{|T_k|=s_k}{=} C_{k+1} \cdot s_k \int_{A_k} h(x)f(x)dx \\
&= C_k \int_{A_k} h(x)f(x)dx.
\end{aligned}$$

Somit folgt die Behauptung aus dem Eindeutigkeitssatz 30.
Damit gliedert sich das Resultat von Knuth in diesen kontinuierliche Fall ein. □

Satz 32 *Für alle j in T_k gilt:*

$$\mu^{(k+1)}(A_{k,j}) = s_k^{-1}. \tag{40}$$

Beweis: (Doberkat (1984))

$$\begin{aligned}
\mu^{(k+1)}(A_{k,j}) &\overset{Satz31}{=} C_{k+1}\mu(A_{k,j}) \\
&\overset{Def.\mu}{=} C_{k+1} \int_{A_{k,j}} f(x)dx
\end{aligned}$$

$$
\begin{aligned}
&\overset{(*)}{=} && C_{k+1}\int_{A_k} f(x)dx \\
&\overset{Def.\mu}{=} && s_k^{-1} C_k \mu(A_k) \\
&\overset{Satz 31}{=} && s_k^{-1}\mu^{(k)}(A_k) \\
&\overset{\mu^{(k)}(A_k)=1}{=} && s_k^{-1}.\Box
\end{aligned}
$$

Die Gleichheit der Umformung $(*)$ gilt nach Satz 29 und Hilfsatz 36, da die Bijektion R_k eine Permutationsmatrix darstellt. Der Betrag der Determinante der Jakobi–Matrix ist folglicherweise gleich 1. $\Box$

Sei $d(k,j)$ die Entfernung von den Elementen zu den Positionen k und j. Sie ist ein Maß für die Anzahl der Vertauschungen, die durchgeführt werden müssen. Sei $f(k,j)$ die Anzahl der Vergleiche für $y \in A_{k,j}$, d.h. $f(1,1) = 2$, $f(k,j) = 0$ für $k > \lfloor n/2 \rfloor$ und

$$f(k,j) = 2(d(k,j) - 1) + s_{\lfloor n/2 \rfloor} + s_j. \tag{41}$$

Nun wird die Erzeugendenfunktion für die Wahrscheinlichkeiten der Anzahl von Vergleichen in T_k entsprechend Satz 32 wie folgt definiert:

$$
\begin{aligned}
\mathcal{V}_k(z) &= s_k^{-1} \cdot \sum_{t=0}^{\infty} \{\mu^{(k+1)}(A_{k,j});\, j \in T_k,\, f(k,j) = t\} \cdot z^t && (42)\\
&= s_k^{-1} \cdot \sum_{t=0}^{\infty} \mid \{j;\, j \in T_k,\, f(k,j) = t\} \mid \cdot z^t. && (43)
\end{aligned}
$$

Satz 33 *Die Erzeugendenfunktion $\mathcal{V}$ für die Wahrscheinlichkeiten der Anzahl von Vergleichen in der Generierungsphase von HEAPSORT ist gegeben durch:*

$$\mathcal{V}(z) = \prod_{k=1}^{n} \mathcal{V}_k(z). \tag{44}$$

Beweis: (Doberkat (1984)) Falls mit $\mathcal{V}_k(z) = \sum_{t_k=0}^{\infty} v_{k,t_k} z^{t_k}$ für $k \in \{1,\ldots,n\}$ eine Menge von Potenzreihen beschrieben wird, so gilt für das n-fache Produkt $(\mid T \mid = t_1 + \ldots + t_n)$:

$$\prod_{k=1}^{n} \mathcal{V}_k(z) = \sum_{t=0}^{\infty} \left(\sum_{\mid T \mid = t} v_{1,t_1} \cdot \ldots \cdot v_{n,t_n} \right) \cdot z^t.$$

Man spricht auch von der n–fachen Faltung der zugehörigen Koeffizientenfolgen. Somit ist:

$$\mathcal{V}(z) = \prod_{k=1}^{n} \frac{s_k \mathcal{V}_k(z)}{s_k}$$

$$
\begin{aligned}
&= C_n \cdot \sum_{t=0}^{\infty} \left(\sum_{|T|=t} \begin{array}{c} |\{j_1; j_1 \in T_1, f(1,j_1) = t_1\}| \\ \cdot \ldots \cdot \\ |\{j_n; j_n \in T_n, f(n,j_n) = t_n\}| \end{array} \right) \cdot z^t \\
&= C_n \cdot \sum_{t=0}^{\infty} \left(\sum_{|T|=t} |\{(j_1, \ldots j_n); j_k \in T_k, \sum_{k=1}^{n} f(k,j_k) = t\}| \right) \cdot z^t \\
&= C_n \cdot \sum_{t=0}^{\infty} \left(\sum_{|T|=t} |\{(j_1, \ldots j_n); j_k \in T_k, \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} f(k,j_k) = t\}| \right) \cdot z^t.
\end{aligned}
$$

Für beliebige $j_k \in T_k$ sei

$$P(j_1, \ldots, j_{\lfloor \frac{n}{2} \rfloor}) = \{x \in A \mid \forall k \in \{1, \ldots, \lfloor \frac{n}{2} \rfloor\} : x^{(k+1)} \in A_{k,j_k}\}.$$

Für alle $x \in P$ wurden $\sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} f(k,j)$ Vergleiche durchgeführt. Für alle Belegungen von $j_1, \ldots, j_{\lfloor \frac{n}{2} \rfloor}$ existiert eine durch eine Permutationsmatrix darstellbare Bijektion von $P(j_1, \ldots, j_{\lfloor \frac{n}{2} \rfloor})$ nach $P(1, \ldots, \lfloor \frac{n}{2} \rfloor)$. Nach dem Transformationssatz gilt demnach

$$\mu(P(j_1, \ldots, j_{\lfloor \frac{n}{2} \rfloor})) = \mu(P(1, \ldots, \lfloor \frac{n}{2} \rfloor)) = C_n^{-1}$$

Damit gilt

$$
\begin{aligned}
& C_n \sum_{t=0}^{\infty} \left(\sum_{|T|=t} |\{(j_1, \ldots j_n); j_k \in T_k, \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} f(k,j_k) = t\}| \right) \cdot z^t \\
= & \sum_{t=0}^{\infty} \left(\sum_{|T|=t} \{\mu(P(j_1, \ldots j_n)); j_k \in T_k, \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} f(k,j_k) = t\} \right) \cdot z^t \\
= & \sum_{t=0}^{\infty} \mu(V = t) \cdot z^t
\end{aligned}
$$

mit V als die Anzahl der Vergleiche in der Aufbauphase. Damit ist $\mathcal{V}(z)$ eine Erzeugendenfunktion für die Anzahl der Vergleiche. □

Da der Mittelwert sich aus der Differentation (mit anschließender Auswertung an der Stelle 1) der zugehörigen Erzeugendenfunktion ergibt, beträgt die mittlere Anzahl von Vergleichen während der Aufbauphase $\mathcal{V}'(1)$. Seien $G(z)$ und $F(z)$ zwei Funktionen mit $G(1) = F(1) = 1$, dann gilt für $H(z) = G(z)F(z)$ nach der Produktregel $H'(z) = G'(z)F(z) + G(z)F'(z)$, also $H'(1) = G'(1) + F'(1)$. Demnach läßt sich die mittlere Anzahl von Vergleichen während der Aufbauphase auch wie folgt bestimmen:

$$\mathcal{V}'(1) = \sum_{k=1}^{n} \mathcal{V}'_k(1). \tag{45}$$

Satz 34 *Sei* $\alpha_i = \sum_{j=1}^{\infty} \frac{1}{2^j-1}$, *d.h.* $\alpha_1 = 1.6066951\ldots$ *bzw.* $\alpha_2 = 1.1373387\ldots$. *Die Generierungsphase von HEAPSORT benötigt im Durchschnitt* $(\alpha_1 + 2\alpha_2 - 2)n + \Theta(logn)$ *Vergleiche und* $(\alpha_1 + \alpha_2 - 2)n + \Theta(\log n)$ *Zuweisungen.*

Beweis:(Skizze nach Doberkat(1984)) Im Fall $n = 2^{l+1} - 1$ lauten die Erzeugendenfunktionen

$$\mathcal{V}_{2^j+i}(z) = \frac{1}{2^{l+1-j}-1} \cdot \frac{2^{l-j} z^{2(n-j)}(3z^2-1) - z^2}{2z^2-1}, \tag{46}$$

wobei $j \in \{0, \ldots, l-1\}$ und $i \in \{0, \ldots, 2^j - 1\}$.
und damit

$$\mathcal{V}(z) = \prod_{j=0}^{l-1} \left(\frac{1}{2^{n+1-j}-1} \cdot \frac{2^{n-j} z^{2(n-j)}(3z^2-1) - z^2}{2z^2-1} \right)^{2^j}. \tag{47}$$

Geschickte logarithmische Differentation liefert mit

$$\begin{aligned} x_l &= \sum_{j=1}^{l} \frac{j}{2^j - 1} \\ y_l &= \sum_{j=1}^{l} \frac{1}{2^j - 1} \end{aligned}$$

folgenden Mittelwert:

$$\mathcal{V}'(1) = 2^{l+1}(2y_{n+1} - x_{n+1} - 2) + 2.\Box \tag{48}$$

Die Erzeugendenfunktion für x_l ist:

$$\mathcal{F}(z) = \frac{1}{1-z} \sum_{k=1}^{\infty} \frac{z}{2^k - z} \tag{49}$$

und für y_l:

$$\mathcal{G}(z) = \frac{z}{1-z} \sum_{k=1}^{\infty} \frac{2^k}{2^k - z} \tag{50}$$

Mit Hilfe von dem Satz von Daboux (Greene (1981)) können die Polstellen der Erzeugendenfunktionen zu einer verfeinerten Approximation von x_l und y_l genutzt werden, was zu den Ergebnissen:

$$\begin{aligned} x_l &= \alpha_1 - \frac{l}{2^l} - \frac{1}{3 \cdot 2^{2l}} + o\left(\frac{1}{2^{2l}}\right) \\ y_l &= \alpha_1 + \alpha_2 - \frac{l}{2^l} - \frac{1}{3 \cdot 2^{2l}} + o\left(\frac{l}{2^{2l}}\right), \end{aligned}$$

wobei α_1 und α_2 die in der Behauptung beschriebenen Konstanten darstellen. Einsetzen in Gleichung 48 liefert dann das zu erzielende Ergebnis.
Im Falle $n \neq 2^{l+1} - 1$ gestalten sich die Formeln für die Erzeugendenfunktionen wesentlich komlexer. Dennoch sind im Grunde die gleichen Schritte durchzuführen, um zu dem behaupteten Ergebnis zu gelangen.
Der Beweis für die Anzahl der Zuweisungen verläuft analog.□

10.2 Aufbauphase von BOTTOM-UP-HEAPSORT

Geduld ist zweierlei: Ruhige Ertragung des Mangels und Ruhige Ertragung des Übermaßes.

Novalis

Die Analyse dieser Phase ist durch die Resultate der HEAPSORT-Untersuchungen zu erschließen:

Satz 35 *Sei* $\alpha_i = \sum_{j=1}^{\infty} \frac{1}{2^j - 1}$, *d.h.* $\alpha_1 = 1.6066951\ldots$ *bzw.* $\alpha_2 = 1.1373387\ldots$, *und* $\beta = \sum_{h=2}^{\infty} \frac{1}{2^h(2^h-1)} \approx 0.1066952$. *BOTTOM-UP-HEAPSORT benötigt für die Aufbauphase im Mittel*

$$(9/2 - \alpha_1 - \alpha_2 - \beta)n + \Theta(\log n) \approx 1.649271n$$

Vergleiche.

Beweis: (Wegener (1993)) Wenn mit l_{HS} die halbe Anzahl an Vergleichen während eines *ReHeap*-Schrittes von HEAPSORT bezeichnet wird und l_{BUS} die Anzahl der Vergleiche von *BottomUpSearch* bezeichnet, so gelten die folgenden Beziehungen:

$$l_{HS} + l_{BUS} = d + 2 \qquad \text{wenn } 0 < j < d,$$

$$l_{HS} + l_{BUS} = d + 1 \qquad \text{wenn } 0 = j \text{ oder } j = d,$$

wobei d die Länge des aktuellen speziellen Pfades ist.
Mit L_{HS}, L_{BUS} und D seien die Zufallsvariablen der Summen aller l_{HS}, l_{BUS} und d bezeichnet. Satz 34 zeigt auf, daß der Mittelwert $E(L_{HS})$ gleich $(\alpha_1/2 + \alpha_2 - 1)n + \Theta(\log n)$ ist. Die Hilfssätze 18 und 20 führen hingegen zu $E(D) = n + \Theta(\log n)$.
Sei T eine Zufallsvariable, die die Aufrufe von *BottomUpSearch* zählt, wo $l_{HS} + l_{BUS} = d + 1$ gilt. Dann ist der gesuchte Erwartungswert:

$$\begin{aligned} E(L_{BUS}) &= E(D) + 2\lfloor n/2 \rfloor - E(T) - E(L_{HS}) \\ &= (3 - \alpha_1/2 - \alpha_2)n - E(T) + \Theta(\log n). \end{aligned}$$

Um $E(T)$ zu bestimmen betrachte zunächst den Fall $j = 0$, d.h. die Wurzel ist das kleinste Element im betrachteten Teilbaum. Wenn in dem Teilbaum

r Elemente liegen, ist die Wahrscheinlichkeit diese Falles gleich $1/r$. Da ein Fehler der Größe $\Theta(\log n)$ zugelassen wird, betrachte vereinfachenderweise, daß in den $n/2^h$ Teilbäume $2^h - 1$ Elemente liegen, d.h. alle betrachteten Teilbäume vollständig sind. Damit liegt die erwartete Anzahl von Situationen, wo $j = 0$ ist, bei $\beta n + \Theta(\log n)$, wobei

$$\beta = \sum_{h=2}^{\infty} \frac{1}{2^h(2^h - 1)} \approx 0.1066952.$$

Betrachte nun den Fall $j = d$, d.h. das kleinste, speziell genannte, Element auf dem speziellen Pfad liegt an einem Blatt an. Sei c die Anzahl von Vergleichen und i die Anzahl von Zuweisungen, die das HEAPSORT-*ReHeap* benötigt. Weiterhin sei s die Anzahl der Kinder des speziellen Elementes. Dann gilt $c = 2i + s$. Es bezeichnen C,S bzw. I die Zufallsvariablen, die die Summe von allen c,s bzw. i beschreiben. Nach Satz 34 gilt: $E(S) = E(C) - 2E(I) = (2 - \alpha_1)n + \Theta(\log n)$. Für maximal $\lfloor \log n \rfloor$ Fälle kann $s = 1$ gelten, da alle Teilbäume zu Wurzeln, die nicht auf dem speziellen Pfad liegen, vollständig sind.
Folglicherweise liegt die Wahrscheinlichkeit, daß $s = 2$ ist, bei

$$\frac{E(S)}{\lfloor n/2 \rfloor} = 2 - \alpha_1 + \Theta\left(\frac{\log n}{n}\right) \approx 0.3933049.$$

Und die Gegenwahrscheinlichkeit 0.6066951 bezeichnet im Rahmen der vorgegebenen Fehlerschranke $\Theta(\log n/n)$ die Wahrscheinlichkeit, daß $s = 0$ gilt.
Damit ist das Mittel der Situation $j = d$:

$$\left(\alpha_1 - 1 + \Theta\left(\frac{\log n}{n}\right)\right) \cdot \lfloor n/2 \rfloor = (\alpha_1/2 - 1/2))n + \Theta(\log n).$$

Somit ergibt sich $E(T)$ aus der Summe der Ergebnisse zu den Fällen $j = 0$ und $j = d$ und letztendlich ist:

$$E(L_{BUS}) = (7/2 - \alpha_1 - \alpha_2 - \beta)n + \Theta(\log n).$$

Die Addition der Vergleichsanzahl, die durch *LeafSearch* gemäß Hilfssatz 20 entsteht, liefert die Behauptung.□

11 Die average-case Analysen der Auswahlphase

Alles beginnt mit der Sehnsucht.

Nelly Sachs

11.1 Die average-case Analyse von BOTTOM-UP-HEAPSORT

Warum scheuen Sie sich auch so sehr, etwas zu wiederholen, das schon vor Ihnen gesagt worden ist. In Verbindung mit Ihren eigenen Gedanken erscheint das Alte selbst doch immer von einer neuen Seite.

Moses Mendelssohn
an Immanuel Kant, 25.12.1770

Die Auswahlphase zerstört die Gleichverteilung. Schon nach der Entnahme eines Elementes an der Wurzel ist die Wahrscheinlichkeit der so erzeugten Heaps untereinander nicht gleich. Der Beleg dazu kann durch ein einfaches Zahlenbeispiel gefunden werden. Es gibt nach der Generierungsphase 3 gleichwahrscheinlich auftretende Heaps der Größe 4. Somit können dann die 2 existierenden Heaps der Größe 3 nach der Entnahme der Wurzel nicht gleichwahrscheinlich sein.
Diese Betrachtung hat eine exakte average-case Analyse von allen HEAPSORT-Varianten bis dato unmöglich gemacht.
Sedgewick und Schaffer (1993) erzielen durch ein vergleichsmäßig einfaches Aufzählungsargument folgendes Ergebnis:

Satz 36 *Die Durchschnittsanzahl von Vergleichen für BOTTOM-UP-HEAPSORT ist kleiner als* $n \log n + n \log\log n + O(n)$.

Beweis: (Wegener (1995)) Der Beweis stützt sich auf die Größen des durch die Rückwärtsanalyse generierten Konfigurationenbaum.
Es reicht zu zeigen, daß die durchschnittliche Summe der Einsinktiefen mindestens $(1-\epsilon^n)(n \log n - n \log\log n - 3n)$ für ein $0 < \epsilon < 1$ beträgt.
Sei $d_1, \ldots, d_n$ eine Folge von möglichen Einsinktiefen, wobei sich d_i auf die *ReHeap*-Prozedur nach der Entfernung des Objektes i bezieht. Seien $j_1, \ldots, j_n$ die korrespondierenden Indizes in der Rückwärtsanalyse, d.h. j_i ist der Index, der für *PullUp*(j_i) gewählt wurde, um den Heap der Größe i nach Entfernen des Objektes i wieder zu rekonstruieren. Demnach entspricht eine *PullUp*–Folge genau einem Heap.
Es kann direkt die Beziehung $d_i = \lfloor \log j_i \rfloor$ abgelesen werden, da das Element i nur bis zur Stelle j_i einsinken wird. Umgekehrt gibt es für jedes d_i nur 2^{d_i} mögliche j_i-Werte. Da $j_1 = 0$, $j_2 = 0$ und $j_i \in \{1, \ldots, \lfloor \log n \rfloor\}$ für $i \geq 3$ ist, gibt es weniger als $\lfloor \log n \rfloor^n$ Möglichkeiten, eine *PullUp*– Folge zu bilden. Weiterhin ist die Zahl der zu $d_1, \ldots, d_n$ korrespondierenden *PullUp*–Folgen kleiner als

$$\prod_{i=1}^{n} 2^{d_i} = 2^{\sum_{i=1}^{n} d_i}.$$

Die Idee ist es, daß Folgen kleiner Einsinktiefen nur wenige zugehörige *PullUp*-Folgen haben. Sei $M = \sum_{i=1}^{n} d_i$. Damit ist die Zahl der $Pull - Up$-Folgen zu Folgen von Einsinktiefen mit $J \leq M$ durch $\lfloor \log n \rfloor^n 2^J$ beschränkt.

Für $J = n(\log n - \log\log n - 3)$ ergibt sich, daß die Anzahl der Heaps, deren Summe der Einsinktiefen durch J beschränkt ist, selber durch

$$(\log n)^n 2^J = 2^{n(\log n + \log\log n - \log\log n - 3)} = \left(\frac{n}{8}\right)^n$$

beschränkt ist.
Die Anzahl aller Heaps ist nach Satz 20 gleich $f(n) = \frac{n!}{\prod_{i=1}^{n} s_i}$, wobei s_i die Größe des Teilbaumes zur Wurzel i ist.
Sedgewick und Schaffer (1993) geben eine untere Grenze für $f(n)$ mit $(n/7)^n$ an.
Nunmehr erweist sich die Wahl von J als geeignet. Wenn die Summe der Einsinktiefen größer als J sind so werden sie mit J bewertet, anderenfalls mit 0. Damit ist der Erwartungswert für die Summe der Einsinktiefen mindestens

$$J \cdot \left(1 - \frac{(n/8)^n}{(n/7)^n}\right) + 0 \cdot \left(\frac{(n/8)^n}{(n/7)^n}\right) = J \cdot \left(1 - \left(\frac{7}{8}\right)^n\right). \square \qquad (51)$$

Li und Vitányi (1993) stellen eine Analyseidee von Ian Munro vor, die zu einem average-case von $n \log n + O(n)$ führt. Sie wird im folgenden unabhängig von der dortigen Grundlage der Kolmogoroff Komplexität dargestellt.

Definition 6 *Eine binäre Codierung einer Menge M ist eine injektive Abbildung $C : M \to \{0,1\}^*$. Die Codierungslänge eines Elementes x sei mit $l_C(x)$ bezeichnet und kennzeichne die Tupelgröße des Bildes von x.*

Satz 37 *(Inkompressionstheorem). Sei $|M| = m$. Für alle Codierungen C und alle $b \in \{0, \ldots, \lceil \log m \rceil\}$ existieren mindestens $m(1 - 2^{-b}) + 1$ Elemente, die eine Codierungslänge von mindestens $\lceil \log m \rceil - b$ haben.*

Beweis: (Li und Vitányi (1993)) Die Anzahl aller Bilder von M unter C ist aufgrund der Injektivität von C gleich m. Die Anzahl aller Darstellungen mit weniger als $\lceil \log m \rceil - b$ Bits ist gleich

$$\sum_{i=0}^{\lceil \log m \rceil - b - 1} 2^i = 2^{\lceil \log m \rceil - b} - 1.$$

Zieht man diese Zahl von der Mächtigkeit der Menge aller Bilder ab, so erhält man die minimale Anzahl der Elemente x, die eine Codierungslänge $l_C(x)$ von mindestens $\lceil \log m \rceil - b$ haben:

$$m - \left(m 2^{\lceil \log m \rceil - b} - 1\right) \geq m\left(1 - 2^{-b}\right) + 1. \square$$

Beispiel 12 *Die Menge S_n aller Permutationen hat nach der Stirling'schen Formel $n! \approx n^n e^{-n}\sqrt{2\pi n}$ viele Elemente (vergl. Anhang). Somit können für alle Codierungen mindestens $1-(1/2)^{n/2}$ Elemente p ausgewählt werden, die eine Codierungslänge $l_C(p)$ von mindestens $n\log n - 2n$ besitzen.*

Hilfssatz 37 *Für alle Codierungen C existieren mindestens $1-(1/2)^{n/2}$ durch die Prozedur Heapify aus einer Permutation $p \in S_n$ generierte Heaps h, so daß die Codierungslänge für h ($l_C(h)$) mindestens $n\log n - 4n$ beträgt.*

Beweis: (Li und Vitányi (1993)[12]) Es wird bei gegenteiliger Annahme gezeigt, daß sonst auch $p \in S_n$ mit weniger als $n\log n - 2n$ codiert werden kann. Dazu wird jeder Pfad P eines einsinkenden Elementes wie folgt codiert: Verzweigt P nach links, so bezeichne die Kante mit den Bitstring '00', verzweigt P nach rechts, so wähle entsprechend '01'. Das Ende eines Pfades wird mit '11' festgelegt. Nach Hilfssatz 18 ist die Summe der Längen der in der Generierung entstehenden Pfade höchstens $n-1$. Damit werden insgesamt weniger als $2n$ Bits zur Codierung der gesamten Aufbauphase benötigt. Die Rückwärtsbetrachtung des Codes ermöglicht es, aus den generierten Heaps durch die somit vollständig festgelegten *PullUp*–Operationen die Ausgangspermutation p aus h zu gewinnen. Damit wäre p mit $n\log n - 4n + 2n = n\log n - 2n$ codiert. Widerspruch. □

Hilfssatz 38 *Sei d_i die Einsinktiefe eines Elementes im i–ten Schritt der Auswahlphase. Dann existiert eine Codierung C' von Heaps h mit den folgenden Eigenschaften:*

1. *Die Länge der Codierung von h beträgt:*

$$l_{C'}(h) = \sum_{i=2}^{n-1} (d_i + 2\lceil \log(\lceil \log n\rceil - d_i + 1)\rceil) \le n\lceil \log n\rceil + 6n.$$

2. *Der Code C' ermöglicht die rückwärige Konstruktion von h mittels PullUp-Folgen.*

Beweis: (Li und Vitányi (1993)[13])

1. Der Pfad P eines einsinkenden Elementes in der Auswahlphase wird wie folgt codiert: Verzweigt P nach links, so bezeichne die Kante mit dem Bitstring '0', verzweigt P nach rechts, so wähle entsprechend '1'. Das Ende von P wird durch die Codierung der Größe $\lceil \log n\rceil - |P|$ festgelegt. Die Trennung von der Größencodierung und des eigentlichen Pfades P wird

[12] Ebd. wird die Grenze $n\log n - 6n$ bewiesen.
[13] Ebd. wird die Ungleichung in 1. nicht gezeigt und die Codierung unötigerweise verkompliziert.

durch eine Verdoppelung der Bitanzahl des Größencodes, einer sogenannten Selbstbeschränkung, erreicht.

Sind mit $d_i, i \in \{2, \ldots, n-1\}$, jeweils die Tiefen der einsinkenden Elemente bezeichnet, so werden zu Codierung aller Pfade P insgeamt

$$l_{C'}(h) = \sum_{i=2}^{n-1} (d_i + 2\lceil \log(\lceil \log n \rceil - d_i + 1) \rceil)$$

Bits benötigt.

Ist $d_i \leq \lceil \log n \rceil - 4$, so gilt $d_i + 2\lceil \log(\lceil \log n \rceil - d_i + 1) \rceil \leq \lceil \log n \rceil + 2$, wie die folgende Betrachtung belegt:

Es existiert ein $x \in \{0, \ldots, \lceil \log n \rceil - 4\}$ mit $d_i = \lceil \log n \rceil - 4 - x$. Damit ist

$$d_i + 2\lceil \log(\lceil \log n \rceil - d_i + 1) \rceil \quad = \quad \lceil \log n \rceil - 4 - x + 2\lceil \log(5 + x) \rceil$$

Für $x \in \{0, 1, 2, 3\}$ stimmt somit die behauptete Ungleichung. Für $x \in \{4, \ldots, \log n - 4\}$ gilt sowohl

$$\begin{aligned} 5 + x < 4x \quad &\Leftrightarrow \quad \log(5 + x) < \log 4 + \log x \\ &\Rightarrow \quad 2\lceil \log(5 + x) \rceil < 2(\lceil \log 4 + \log x \rceil) \\ &\Rightarrow \quad 2\lceil \log(5 + x) \rceil < 2\lceil \log 4 \rceil + 2\lceil \log x \rceil \end{aligned}$$

als auch $x \geq 2\lceil \log x \rceil - 2$ und somit gilt:

$$\begin{aligned} d_i + 2\lceil \log \lceil \log n \rceil - d_i + 1 \rceil \quad &< \quad \lceil \log n \rceil - 4 - x + 2\lceil \log 4 \rceil + 2\lceil \log x \rceil \\ &= \quad \lceil \log n \rceil - x + 2\lceil \log x \rceil \\ &\leq \quad \lceil \log n \rceil + 2. \end{aligned}$$

Falls $\lceil \log n \rceil - 4 < d_i \leq \lceil \log n \rceil$ ist, so gilt:

$$\begin{aligned} d_i + 2\lceil \log(\lceil \log n \rceil - d_i + 1) \rceil \quad &\leq \quad d_i + 2\lceil \log 5 \rceil \\ &\leq \quad \lceil \log n \rceil + 6. \end{aligned}$$

Damit gilt insgesamt:

$$l_{C'}(h) = \sum_{i=2}^{n-1} (d_i + 2\lceil \log(\lceil \log n \rceil - d_i + 1) \rceil) \leq n\lceil \log n \rceil + 6n.$$

2. Die rückwärtige Konstruktion von h aus dem einzigen Heap der Größe 1 ist analog zur Aufbauphase durch die gespeicherten Pfade in C' festgelegten *PullUp*–Operationen in $l_{C'}(h)$ Schritten möglich. □

Hilfssatz 39 *Sei $f(n) \in \omega(1)$. Wenn $\sum_{i=2}^{n-1} d_i = n\lceil \log n \rceil - nf(n)$ gilt, dann ist*

$$nf(n) - \max_{d_i \in \{0,\ldots,\lceil \log n \rceil\}} \left\{ \sum_{i=2}^{n-1} 2\lceil \log(\lceil \log n \rceil - d_i + 1) \rceil \right\} = \omega(n).$$

Beweis: Aufgrund der Konkavität der log–Funktion werden die Extremwerte der in den Variablen d_i definierten Funktion

$$g(d_2, \ldots, d_{n-1}) = \sum_{i=2}^{n-1} 2\lceil \log(\lceil \log n \rceil - d_i + 1) \rceil$$

dort erreicht, wo alle d_i den gleichen Wert haben (Fall 1) oder dort, wo möglichst viele d_i extrem klein bzw. groß gewählt werden (Fall 2).

Fall 1: Die Werte von d_i betragen nach Voraussetzung $(n\lceil \log n \rceil - nf(n))/n = \lceil \log n \rceil - f(n)$. Damit ist der Funktionswert von $nf(n) - g(d_2, \ldots, d_{n-1})$ gleich

$$\begin{aligned} nf(n) - \sum_{i=2}^{n-1} 2\lceil \log(f(n) + 1) \quad &\leq \quad nf(n) - 2n\lceil \log(f(n) + 1) \rceil \\ &= \quad n(f(n) - 2\lceil \log(f(n) + 1) \rceil). \end{aligned}$$

Es gibt ein n_0, so daß für alle $n \geq n_0$

$$2\lceil \log(f(n) + 1) \rceil \leq f(n)/2$$

gilt. Da $f(n) \in \omega(1)$ ist, gilt somit auch $f(n) - 2\lceil \log(f(n) + 1) \rceil \in \omega(1)$ und $nf(n) - g(d_2, \ldots, d_{n-1}) = \omega(n)$.

Fall 2: Die Anzahl der auf $\lceil \log n \rceil$ setzbaren Werte ist höchstens $(n\lceil \log n \rceil - nf(n))/\lceil \log n \rceil = n(1 - (f(n)/\lceil \log n \rceil))$. Dementsprechend sind die auf 0 setzbaren d_i–Werte höchstens $n(f(n)\lceil \log n \rceil)$. Damit ist der Funktionswert von $nf(n) - g(d_2, \ldots, d_{n-1})$ gleich

$$nf(n) - \sum_{i=2, d_i=0}^{n-1} 2\lceil \log(\lceil \log n \rceil + 1) \rceil$$

$$\begin{aligned} &= \quad nf(n) - \frac{2nf(n)\lceil \log(\lceil \log n \rceil + 1) \rceil}{\lceil \log n \rceil} \\ &\leq \quad n\left(f(n) - \frac{2f(n)\lceil \log(\lceil \log n \rceil + 1) \rceil}{\lceil \log n \rceil}\right). \end{aligned}$$

Es gibt ein n_0, so daß für alle $n \geq n_0$

$$\frac{2\lceil\log(\lceil\log n\rceil + 1)\rceil}{\lceil\log n\rceil} \leq 1/2$$

gilt. Da $f(n) \in \omega(1)$ ist, gilt somit auch $f(n) - \frac{2f(n)\lceil\log(\lceil\log n\rceil+1)\rceil}{\lceil\log n\rceil} \in \omega(1)$ und $nf(n) - g(d_2, \ldots, d_{n-1}) = \omega(n)$.□

Satz 38 *Die Durchschnittsanzahl von Vergleichen für BOTTOM-UP-HEAP-SORT ist höchstens als* $n\log n + O(n)$.

Beweis: (Li und Vitányi (1993)[14]) Es reicht zu zeigen, daß die durchschnittliche Summe der Einsinktiefen d_i in der Auswahlphase und ausreichend vielen Fällen mindestens $n\log n - O(n)$ ist.
Nach Hilfssatz 37 sind für einen Mindestanteil von $(1-(1/2)^{n/2})$ der Elemente $p \in S_n$ die Codierungslängen der aus p generierten Heaps h mindestens $n\log n - 4n$. Dieses gilt insbesondere für die in Hilfssatz 38 beschriebenen Codierung C'. Für dessen Codelänge

$$l_{C'}(h) = \sum_{i=2}^{n-1}(d_i + 2\lceil\log(\lceil\log n\rceil - d_i + 1)\rceil)$$

gilt somit

$$l_{C'}(h) = n\log n - \Theta(n). \tag{52}$$

Der Hilfssatz 39 schafft nun eine Verbindung zwischen der Summe der Einsinktiefen $d(n) = \sum_{i=2}^{n-1} d_i$ und der Codierung $l_{C'}(h)$. Falls $d(n)$ superlinear unterhalb von $n\lceil\log n\rceil$ liegt, d.h $d(n) = n\lceil\log n\rceil - f(n)n$ mit $f \in \omega(1)$, dann ergibt sich ein Widerspruch zu Gleichung 52:

$$\begin{aligned} l_{C'}(h) &= \sum_{i=2}^{n-1}(d_i + 2\lceil\log(\lceil\log n\rceil - d_i + 1)\rceil) \\ &\leq n\lceil\log n\rceil - nf(n) + \\ &\quad \max_{d_i\in\{0,\ldots,\lceil\log n\rceil\}}\left\{\sum_{i=2}^{n-1}(d_i + 2\lceil\log(\lceil\log n\rceil - d_i + 1)\rceil)\right\} \\ &= n\lceil\log n\rceil - \omega(n). \end{aligned}$$

Somit ist für eine Konstante c und für einen Anteil von $1-(1/2)^{n/2}$ aller Permutationen und somit aller Heaps die Summe der Einsinktiefen größer als $n\log n - O(n)$. □
Die Eingrenzung von dem Vorfaktor des linearen Terms ist nur unter realistischen Modellannahmen gezeigt worden. Zentral ist der Begriff des *kurzen bzw.*

[14] Ebd. wird der Übergang von der Codierungs- zur Pfadlänge nicht ersichtlich.

langen Weges. Ein (spezieller) Weg ist kurz (bzw. lang), wenn er auf einem Blatt des vorletzten (letzten) Level endet.
Carlsson (1987a) verfolgte den Ansatz, daß in jedem Schritt des Algorithmus zufällige Heaps entstehen, für die die folgende Eigenschaft wesentlich ist:

(M) Wenn an einem Knoten x der linke Teilbaum i und der rechte Teilbaum j Knoten besitzt, so ist die Wahrscheinlichkeit, daß der spezielle Pfad den linken Teilbaum wählt, gleich $i/(i+j)$

Für $n = 2^k - 1$ ergeben sich durch (M) nach (fehlerhaften Berechnungen von) Carlsson $0.279n$ kurze Wege, was von den Experimenten weit entfernt liegt, da diese ca. $0.5n$ kurze Wege versprechen.
Wegener (1993) schlug ein anderes Modell vor:

(M') Die Wahrscheinlichkeit, daß der spezielle Pfad den linken Teilbaum an einem Knoten x wählt, verändert sich während der Auswahlphase nicht.

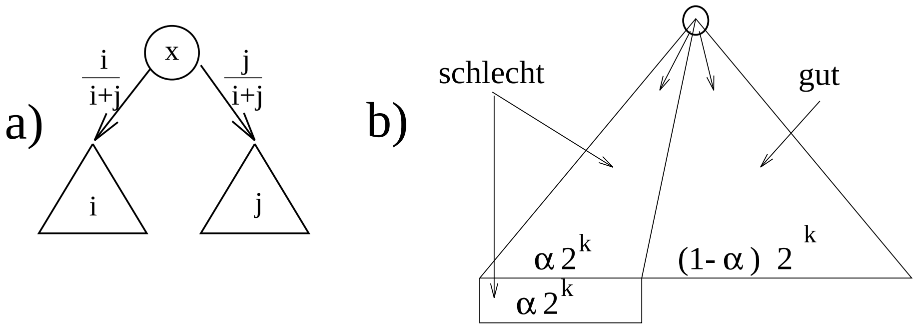

Abbildung 24: Unterschiedlichen Modellannahmen: a) Model (M), b) Model (M').

Es werden die Konsequenzen analysiert, die sich aus (M') ergeben. Sei $n = 2^k - 1 + \alpha 2^k$ mit $0 \le \alpha < 1$. Zum Beginn der Auswahlphase ist die Wahrscheinlichkeit, in die *schlechte* Region zu gelangen gleich $\frac{\alpha 2^k + \alpha 2^k}{\alpha 2^k + \alpha 2^k + (1-\alpha) 2^k} = 2\alpha/(1+\alpha)$, da die Heaps gleichverteilt sind und damit als zufällig angenommen werden können. Entprechend ist die Wahrscheinlichkeit für die *gute Region* $(1-\alpha)/(1+\alpha)$.
Falls in den ersten $\alpha 2^k$ Aufrufe von $BottomUpReHeap$ die gute Region betreten wird, so ist der spezielle Weg kurz. Wird die schlechte Region betreten, so sind eventuell einige Blätter schon eliminiert und damit ist der spezielle Weg nicht unbedingt lang. Im Mittel sind die Hälfte der Wege lang. Damit ist die erwartete Anzahl von kurzen Wegen:

$$\left(1 \frac{1-\alpha}{1+\alpha} + \frac{1}{2} \cdot \frac{2\alpha}{1+\alpha}\right) \alpha 2^k = \frac{\alpha}{1+\alpha} 2^k \qquad (53)$$

Die schlechte Region wird für die folgenden $2^k - 1 \approx 2^k$ *BottomUpReHeap* Aufrufe im Mittel $(2\alpha/(1+\alpha))2^k$–fach betreten, während die gute Region entsprechend $((1-\alpha)/(1+\alpha))2^k$ mal erreicht wird.
Wenn der Baum auf dem letzen Level (Tiefe d) mehr als $\alpha 2^d$ Knoten besitzt, so sind alle Wege in der schlechten Region lang und in der guten Region zur Hälfte kurz. Somit ergeben sich in diesem Fall insgesamt

$$\frac{1}{2}\alpha \cdot \frac{2\alpha}{1+\alpha}2^k = \frac{\alpha^2}{1+\alpha}2^k \tag{54}$$

kurze Wege.
Wenn der Baum auf dem letzen Level (Tiefe d) weniger als $\alpha 2^d$ Knoten besitzt so sind alle Wege in der guten Region kurz und in der schlechten Region zur Hälfte lang. Somit ergeben sich in diesem Fall insgesamt

$$\left(\frac{1}{2}(1-\alpha)+\alpha\right) \cdot \left(\frac{1-\alpha}{1+\alpha}2^k\right) = \frac{1}{2}\frac{1-\alpha^2}{1+\alpha}2^k \tag{55}$$

kurze Wege.
Faßt man die drei Ergebnisse zusammen erhält man letztendlich

$$\frac{1}{1+\alpha}(\alpha+\alpha^2+1/2-\alpha^2/2)2^k = \frac{1}{1+\alpha}(\alpha^2+2\alpha+1)2^{k-1} = (1+\alpha)2^{k-1}$$

$$\approx (1+\alpha)2^{k-1} - 1/2 = n/2.$$

kurze Wege, was den experimentellen Ergebnissen wesentlich besser entspricht. Die Anzahl der kurzen Wege ist für $n \approx 2^k$ und für $n > 2000$ zwischen $0.469n$ und $0.471n$ und für $n \approx 1.4 \cdot 2^k$ ungefähr $0.519n$. Die übrigen Werte liegen innerhalb dieser Grenzen. Wegener (1993) erklärt diese Abweichungen durch ein erweitertes (realistischeres) Modell, das die Wartezeiten der Elemente bis zu einer Vertauschung mit einbezieht.
Doberkats (1982) Untersuchungen zur Entnahme einer Wurzel lassen erkennen, daß die Anzahl der Vergleiche für *BottomUpSearch* während der Auswahlphase für jeweils als zufällige angenommende Heaps $1.299n$ beträgt.
Die Modellannahme (M') (vergl. Wegener (1993)) führt auch zu einer Begründung, warum in der Auswahlphase wesentlich weniger Vergleiche für *BottomUpSearch* als für zufällige Heaps benötigt werden. Dennoch läßt sie die nahezu konstante Anahl von $[1.169, 1.171]n$ Vergleichen nicht schlußfolgern.

Vermutung 1 *Sei $d(n)$ so gewählt, daß $n \log n + d(n)n$ die erwartete Anzahl an Schlüsselvergleichen von BOTTOM-UP-HEAPSORT ist. Dann liegt $d(n)$ im Intervall von [0.34,0.39]. Diese Zahl ist groß für $n \approx 2^k$ und klein für $n \approx 1.4 \cdot 2^k$.*

Begründung: Die Anzahl der Vergleiche für die Prozedur *LeafSearch* in der Auswahlphase beträgt $n \log n - c(n)$ minus der erwarteten Anzahl der kurzen

Wege. Die erwarteten Vergleiche für die Aufrufe von *BottomUpSearch* liegen zwischen $1.169n$ und $1.171n$. Addiert man Durchschnittsergebnis der Generierungsphase von approx. $1.649271n$ hinzu wird für $d(n)$ das folgende Resultat erreicht:

$$-[1.91393, 2] - [0.469, 0.519] + [1.169, 1.171] + 1.649271 = [0.22927, 0.437241].$$

Experimente belegen, daß die Beispiele nicht aus dem Intervall [0.34,0.39] fallen.

11.2 Die average-case Analyse von WEAK-HEAPSORT

Die Schwierigkeiten wachsen, je näher man dem Ziele kommt.

Johann Wolfgang von Goethe

In der average-case Analyse von HEAPSORT bestand die Grundidee darin, durch die Rückwärtsanalyse die starke Variation der Einsinktiefen zu kontrollieren. Zu Folgen geringer Einsinktiefen gab es nur wenige korrespondierende Codierungs- bzw. *PullDown*-Folgen und somit wenig rückwärts generierte Heaps. Die Einsinktiefen waren ein direkter Maßstab für die Anzahl der Vergleiche.

Im WEAK-HEAPSORT Algorithmus ist der Unterschied der Vergleichsanzahl zwischen dem günstigsten und ungünstigsten Fall maximal 1. Es gibt demnach keine *Ausreißer* von Einsinktiefen. So genügt es, anstatt eine Folge von Einsinktiefen, eine Folge von Fällen a), b) bzw. c) (vergl. Abb. 8) zu betrachten. Dieser Folge sollen dann korrespondierende *MergeForestUp*-Folgen zugeordnet werden.

Eine Vereinfachung der Notation wird durch die Zusammenfassung von dem günstigen Fall b) und dem günstigen Fall c) erreicht. Dazu sei $P = \{\lfloor (n-1)/2^i \rfloor \mid i \in \{0,\ldots,k\}\}$ der Weg von Index $n-1$ zurück zum Index 1, wobei $k = \lfloor \log(n-1) \rfloor$ ist.

Ist $n = 2^k - 1$, so wird mittels $Swap(0,n)$ nach *MergeForestUp* ein Element auf den nächsten Level befördert. Der rückwärtig generierte Pfad hat maximale Länge, doch es tritt nach dem Wurzeltausch der günstige Fall an Vergleichen in *MergeForest* auf.

Die vereinfachte Fallunterscheidung lautet:

Fall(I) Es ist $n+2^k-1$ oder der Weg P besitzt die maximale Länge $\lceil \log(n+1) \rceil$.

Fall(II) Der Weg P besitzt die minimale Länge $\lceil \log(n+1) \rceil - 1$.

Im Fall (II) benötigt der Algorithmus garantiert die günstige Zahl von $\lceil \log(n+1) \rceil - 1$ Vergleichen. Ziel wird es sein, die erwartete Anzahl von Situationen, in denen Fall (II) eintrifft, nach unten abzuschätzen und von der worst-case Anzahl an Vergleichen abzuziehen.

Definition 7 *Sei ein Weak-Heap der Größe n gegeben und seien alle Reversebits auf 0 gesetzt. Sei $n-1=(b_k\dots b_0)_2$ mit $k=\lfloor\log(n-1)\rfloor$ und $P_n=\{\lfloor(n-1)/2^i\rfloor \mid i\in\{0,\dots,k\}\}=\{(1\ \ b_{k-1}\dots b_{k-j})_2 \mid i\in\{0,\dots,k\}\}$. Die Menge R_n (bzw. L_n) ist die Menge der Indizes, die rechts (bzw. links) von P_n liegen.*

Per Definition sind die Indizes der Elemente, die in R_n und L_n liegen, unabhängig von der Belegung der Reversebits und somit nur durch n bestimmt.

Hilfssatz 40

$$\begin{aligned}|R_n| &= 2^{k+1}-(k+2)-\sum_{j=0}^{k-1}(1\ \ b_{k-1}\dots b_{k-j})_2,\\ |L_n| &= \sum_{j=0}^{k}(1\ \ b_{k-1}\dots b_{k-j})_2-2^{k+1}+1,\end{aligned}$$

Beweis: Es wird zur Berechnung von $|R_n|$ das Pfadelement von der Levelobergrenze subtrahiert und über die bestehenden Level summiert.

$$\begin{aligned}|R_n| &= \sum_{j=0}^{k-1}\left(2^{j+1}-1-(1\ \ b_{k-1}\dots b_{k-j})_2\right)\\ &= 2\sum_{j=0}^{k-1}2^j-\sum_{j=0}^{k-1}1-\sum_{j=0}^{k-1}(1\ \ b_{k-1}\dots b_{k-j})_2\\ &= 2(2^k-1)-k-\sum_{j=0}^{k-1}(1\ \ b_{k-1}\dots b_{k-j})_2.\end{aligned}$$

Entsprechend wird zur Berechnung von $|L_n|$ die Leveluntergrenze vom Pfadelement subtrahiert und über die bestehenden Level summiert:

$$\begin{aligned}|L_n| &= \sum_{j=0}^{k}\left((1\ \ b_{k-1}\dots b_{k-j})_2-2^j\right)\\ &= \sum_{j=0}^{k}(1\ \ b_{k-1}\dots b_{k-j})_2-\sum_{j=0}^{k}2^j\\ &= \sum_{j=0}^{k}(1\ \ b_{k-1}\dots b_{k-j})_2-2^{k+1}+1.\square\end{aligned}$$

Die zentrale Eigenschaft der Mengen R_n, L_n und P_n wird in dem folgenden Satz beschrieben.

Sei dazu die Menge S_j wieder wie folgt festgelegt:

$$y \in S_j :\Longleftrightarrow Gparent(y) = j$$

bzw.
$S_j = \{rchild(j)\} \cup \{y \mid y \text{ ist von } rchild(j) \text{ nur über linke Kinder erreichbar}\}$.

Satz 39 *Sei j so gewählt, daß $MergeForestUp(j)$ aufgerufen wird.*
Ist $j \in R_n$, dann wird ein Weak-Heap generiert, dessen Pfad P_{n+1} minimale Länge hat, d.h. es tritt Fall(II) ein.
Ist $j \in L_n$, so wird ein Weak-Heap generiert, dessen Pfad P_{n+1} maximale Länge hat, d.h. es tritt Fall(I) ein.
Ist $j \in P_n \cup \{0\}$ und liegt ein (kein) Element von $S_j \cup \{j\}$ auf dem letzten Level, so wird ein Weak-Heap generiert, dessen Pfad P_{n+1} maximale (minimale) Länge hat.

Beweis: Um mittels $MergeForestUp(j)$ ein Element an die Wurzel zu bewegen, muß es im vorletzten Schritt auf dem Pfad P_{n+1} des zu generierenden Weak-Heaps liegen. Die durch j beschriebenen Teilbäume sind für $j \in R_n \cup L_n$ vollständig, da j nicht auf P_n liegt. Der letzte Tausch von den Werten an den Positionen j und 0 verändert somit nicht die Struktur des Weak-Heaps. Ist $j \in R_n$, so ist die Tiefe der Teilbäume, die durch j beschrieben werden, um 1 geringer als im Fall $j \in L_n$. Damit hat P_{n+1} im ersten Fall minimale und im zweiten Fall maximale Länge. Ist $j \in P_n$, so wird S_j im letzten Tausch zum Teil–P_{n+1} des zu generierenden Weak-Heaps und demnach überträgt sich die Maximalität (Minimalität) der Pfadlänge. Für den Sonderfall $j = 0$ wird kein Tausch durchgeführt, doch es gilt $S_j = P_{n+1}$. Ist $j = n - 1$, so liegt der Index auf dem letzten Level wird zum letzten Element auf P_{n+1}. □

Beispiel 13 *Sei $n = 11$. Dann ist $R_n = \{3, 6, 7\}$, $L_n = \{4, 8, 9\}$, $P_n = \{1, 2, 5, 10\}$.*
Die Knoten der Indizes, die zu Fall I) führen, sind in Abb. 25 weiß, die zu Fall II) führen, schwarz gefärbt.

Algorithmisch läßt sich die Klassifizierung der zu generierenden Fälle (Fall(I) = TRUE, Fall(II) = FALSE) wie folgt formulieren:

```
PROCEDURE FALL(n,j):BOOLEAN
  IF n=2^k-1 THEN RETURN(FALSE)
  d = depth(j)
  k = depth(n-1)
  IF (n-1) DIV 2^(k-d) > j      {  j ist in Ln   }
  THEN RETURN(FALSE)
  FI
  IF (n-1) DIV 2^(k-d) < j      {  j ist in Rn   }
```

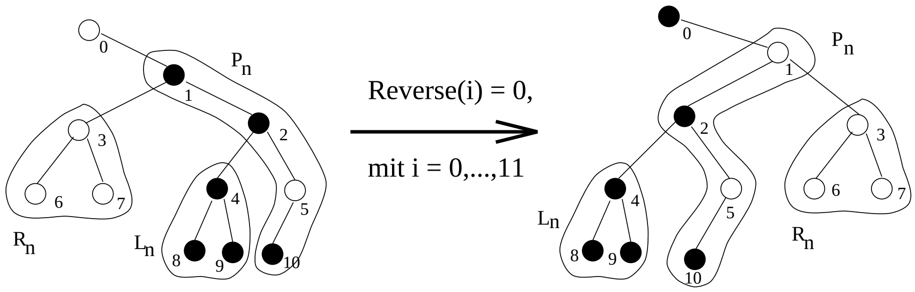

Abbildung 25: Die Lage der Mengen R_n, L_n und P_n.

```
  THEN RETURN(FALSE)
  FI                              {  j ist nun in Pn }
  IF j = n-1                      {  Sonderfall:    }
  THEN FALL = TRUE; EXIT          {  j letzte Elem. in Pn }
  FI
  x = 2 * j + 1 - Reverse[j]
  IF x > n-1 THEN                 {  Sonderfall    }
  THEN FALL = FALSE; EXIT         {  j vorletzte Elem. in Pn }
  FI
  WHILE 2*x+Reverse[x] < n DO
    x = 2*x + Reverse[x];
  OD;
  IF x < 2^(k-1)
  THEN FALL = FALSE               {  S_j  minimal  }
  ELSE FALL = TRUE                {  S_j  maximal  }
  FI
END FALL.
```

Die Mengen $|R_n|$ und $|L_n|$ wachsen unregelmäßig:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\mid R_n \mid$	0	0	0	0	1	1	0	0	4	4	3	3	1	1	0	0	11
$\mid L_n \mid$	0	0	0	1	0	1	3	4	0	1	3	4	7	8	10	11	0
$\mid P_n \mid$	0	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	5

Tabelle 3: Ausschnitt der Funktionstabelle für $|R_n|$, $|L_n|$ und$|P_n|$.

Offensichtlich gilt: $|R_n|+|L_n|+|P_n| = n-1$. Da $|P_n| = \lceil \log n \rceil = \lfloor \log(n-1) \rfloor + 1$ ist, reicht es, eine der Größen $|R_n|$ und $|L_n|$ genauer zu analysieren.

Hilfssatz 41 *Sei $2^k + 1 \leq n \leq 2^{k+1}$, d.h. $n = 2^k + m$ mit $m \in \{1, \ldots, 2^k\}$.*

Dann gilt für alle $m \in \{1, \ldots, 2^k\}$ folgende Symmetrieeigenschaft:

$$|R_{2^k+m}| + |R_{2^{k+1}+1-m}| = |R_{2^k+1}| = 2^k - (k+1)$$

Beweis: Sei $b^m = (b_k^m \ldots b_0^m)_2 = (2^k + m) - 1$ und $c^m = (c_k^m \ldots c_0^m)_2 = (2^{k+1} + 1 - m) - 1$. Es gilt $b_k^m = 1 = c_k^m$. Damit ist

$$(1\ \ b_{k-1}^m \ldots b_{k-j}^m)_2 + (1\ \ c_{k-1}^m \ldots c_{k-j}^m)_2 = \lfloor \frac{2^k + 2^{k+1} - 1}{2^{k-j}} \rfloor. \tag{56}$$

Die Anwendung von Hilfssatz 40 und Gleichung 56

$$\begin{aligned}
|R_{2^k+m}| + |R_{2^{k+1}+1-m}| &= 2^{k+1} - (k+2) - \sum_{j=0}^{k-1} (1\ \ b_{k-1}^m \ldots b_{k-j}^m)_2 + \\
&\quad 2^{k+1} - (k+2) - \sum_{j=0}^{k-1} (1\ \ c_{k-1}^m \ldots c_{k-j}^m)_2 \\
&= 2^{k+2} - 2(k+2) - \sum_{j=0}^{k-1} \lfloor \frac{2^k + 2^{k+1} - 1}{2^{k-j}} \rfloor \\
&= 2^{k+2} - 2(k+2) - \sum_{j=0}^{k-1} \lfloor 2^j + 2^{j+1} - \frac{1}{2^{k-j}} \rfloor \\
&= 2^{k+2} - 2(k+2) - 2^k - 2^{k+1} + 3 - \sum_{j=0}^{k-1} \lfloor -\frac{1}{2^{k-j}} \rfloor \\
&= 2^k - k - 1.
\end{aligned}$$

Für $m = 1$ ist $|R_{2^{k+1}+1-m}| = 0$ und damit $|R_{2^{k+1}+1}| = 2^k - k - 1$. □
Nun läßt sich ermitteln, wie stark die Menge R_n im Mittel wächst:

Satz 40 *Sei $n = 2^k$.*

$$\frac{\sum_{i=1}^n |R_i|}{n} = \frac{n}{6} - \frac{k-1}{2} - \frac{2}{3 \cdot n}. \tag{57}$$

Beweis:

$$\begin{aligned}
\frac{\sum_{i=1}^n |R_i|}{n} &= \frac{1}{n} \sum_{j=1}^{k-1} \sum_{m=1}^{2^j} |R_{2^j+m}| \\
&= \frac{1}{2n} \sum_{j=1}^{k-1} 2^j |R_{2^j+1}| \\
&= \frac{1}{2n} \sum_{j=1}^{k-1} 2^j (2^j - (j+1))
\end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{2n}\sum_{j=1}^{k-1}\left(4^j - j2^j - 2^j\right)\\
&= \frac{1}{2n}\sum_{j=0}^{k-1}\left(4^j - j2^j - 2^j\right)\\
&= \frac{1}{2n}\left\{\frac{4^k-1}{3} - ((k-2)2^k+2) - (2^k-1)\right\}\\
&= \frac{1}{2n}\left\{\frac{n^2-1}{3} - (k-1)2^k - 1\right\}\\
&= \frac{1}{2n}\left\{\frac{n^2}{3} - (k-1)n - \frac{4}{3}\right\}\\
&= \frac{n}{6} - \frac{k-1}{2} - \frac{2}{3\cdot n}.\Box
\end{aligned}
$$

Um die average-case Analyse durchzuführen, betrachte folgendes Modell:

(M) Die Weak-Heaps der Größe n seien in allen Schritten der Auswahlphase gleichverteilt, d.h. jeder zulässige Weak-Heap tritt im Algorithmus mit der gleichen Wahrscheinlichkeit auf.

Das Modell (M) beschreibt nicht die Wirklichkeit, wie die Untersuchungen der Auswahlphase mittels der Rückwärtsanalyse belegen, doch es kann angenommen werden, daß (M) durch das Ausgleichsbestreben der Verteilungen bei großen n annähernd gilt.

Definition 8 *Seien alle $i!$ paarweise verschiedene eingebettete Weak-Heaps der Größe i gegeben. In der Rückwärtsanalyse bezeichne $F(i,j)$ die Anzahl der zulässig zu generierenden Weak-Heaps der Größe $i+1$, die durch $MergeForestUp(j)$ gebildet werden. Dabei sei das Reversebit an der Position $i+1$ auf einen Wert festgelegt. Desweiteren sei*

$$f(i,j) = \frac{F(i,j)}{\frac{(i+1)!}{2}} \tag{58}$$

der Anteil dieser Weak-Heaps unter allen zulässig zu generierenden Weak-Heaps der Größe $i+1$.

Folgerung 15 *Unter dem Modell (M) beschreibt $f(i,j)$ die Wahrscheinlichkeit, daß unter allen zulässig zu generierenden Weak-Heaps der Größe $i+1$ der Index j für $MergeForestUp(j)$ gewählt wurde.*

Für kleine i lassen sich die Werte $f(i,j)$ direkt berechnen (vergl. Tabelle 4).

j/i	2	3	4	5	6	7	8	9	10
0	$\frac{1}{3}$	$\frac{1}{6}$	$\frac{1}{10}$	$\frac{1}{15}$	$\frac{1}{21}$	$\frac{1}{28}$	$\frac{1}{36}$	$\frac{1}{45}$	$\frac{1}{55}$
1	$\frac{2}{3}$	$\frac{1}{2}$	$\frac{7}{30}$	$\frac{7}{45}$	$\frac{16}{105}$	$\frac{41}{420}$	$\frac{13}{180}$	$\frac{13}{225}$	$\frac{8}{165}$
2	0	$\frac{1}{3}$	$\frac{2}{5}$	$\frac{1}{3}$	$\frac{2}{15}$	$\frac{7}{60}$	$\frac{7}{54}$	$\frac{7}{75}$	$\frac{16}{165}$
3	0	0	$\frac{4}{15}$	$\frac{2}{9}$	$\frac{2}{7}$	$\frac{1}{4}$	$\frac{28}{270}$	$\frac{7}{75}$	$\frac{14}{165}$
4	0	0	0	$\frac{2}{9}$	$\frac{4}{21}$	$\frac{1}{6}$	$\frac{2}{9}$	$\frac{1}{5}$	$\frac{14}{165}$
5	0	0	0	0	$\frac{4}{21}$	$\frac{1}{6}$	$\frac{4}{27}$	$\frac{2}{15}$	$\frac{2}{11}$
6	0	0	0	0	0	$\frac{1}{6}$	$\frac{4}{27}$	$\frac{2}{15}$	$\frac{4}{33}$
7	0	0	0	0	0	0	$\frac{4}{27}$	$\frac{2}{15}$	$\frac{4}{33}$
8	0	0	0	0	0	0	0	$\frac{2}{15}$	$\frac{4}{33}$
9	0	0	0	0	0	0	0	0	$\frac{4}{33}$

Tabelle 4: Werte für $f(i,j)$, wobei die Zeilen mit i, die Spalten mit j bezeichnet sind.

Hilfssatz 42 *Sei j der Index eines Blattes ungleich $\lfloor i/2 \rfloor$ in einem Weak-Heap der Größe i. Dann gilt:*

$$f(i,j) = \frac{4}{3(i+1)}. \tag{59}$$

Beweis: Da $j \neq \lfloor i/2 \rfloor$ ist, gilt $Gparent(i) \neq Gparent(j)$ (Vergl. Hilfssatz 6). Definiere die folgenden Ereignisse, die nach der Generierung eines Weak-Heaps der Größe i gelten sollen:

$$\begin{aligned} A_k &= \{WH \text{ mit } a[Gparent(i)] = k \mid WH \text{ ist Weak-Heap der Größe } i\}, \\ B_l &= \{WH \text{ mit } a[j] = l \mid WH \text{ ist Weak-Heap der Größe } i\}. \end{aligned}$$

Sei $q(l,k) = Prob(A_k \cap B_l)$. Nach der Definition der bedingten Wahrscheinlichkeit gilt:

$$q(l,k) = Prob(A_k) \cdot Prob(B_l \mid A_k). \tag{60}$$

Nach Satz 25 ist die Wahrscheinlichkeit $Prob(A_k) = Prob(a[Gparent(i)] = k)$ in einem Weak-Heap der Größe i gleich $1/i$.

Um $Prob(B_l \mid A_k)$ zu berechnen, muß der Wert von $a[Gparent(i)]$ in der Generierung des Heaps festgehalten und dann die Wahrscheinlichkeit $Prob(B_l)$ berechnet werden.

Sei $S = \{(x,y) \in \{1,\ldots,i\}^2 \mid x \neq y, x \neq k, y \neq k\}$ die Menge von Paaren gleichwahrscheinlicher Werte für $a[j]$ und $a[Gparent(j)]$, wenn vor dem Aufbau des Heaps $a[Gparent(i)]$ mit dem Wert k festgelegt wird. (vergl. Hilfssatz 35) Dabei ist j der Index des Blattes ungleich $\lfloor i/2 \rfloor$. An einem solchen Blatt j wird in der Generierung nur ein Vergleich mit $Gparent(j) \neq Gparent(i)$ gemacht. In S sind die Elemente gleichverteilt und es gilt $|S| = 2\binom{i}{2} = (i-1)(i-2)$ (vergl. Abb. 26).

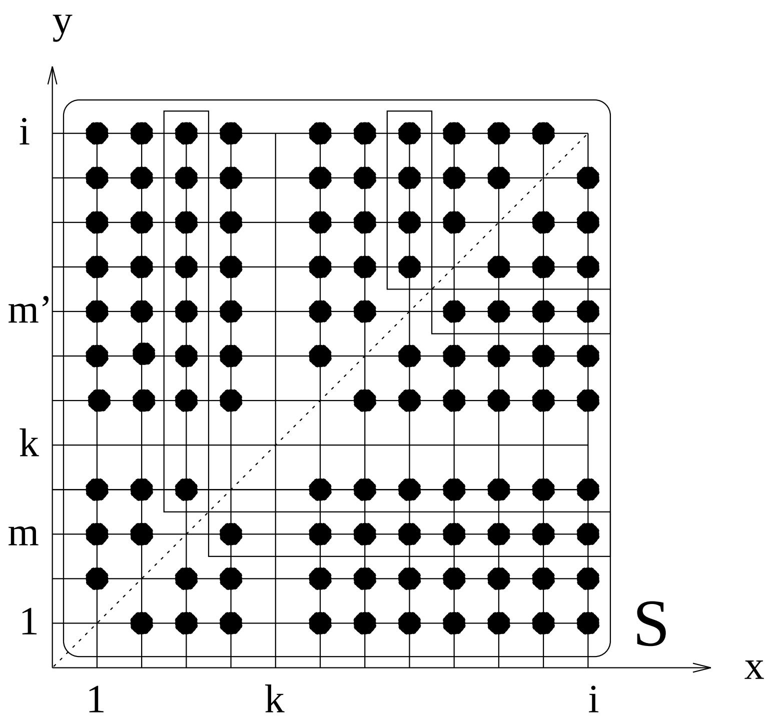

Abbildung 26: Die Lage von k, m und i in der Menge S.

Demnach berechnen sich die Werte $Prob(min\{x,y\} = l)$ als Anteile der günstigen Möglichkeiten, d.h. wo $y = l$ bzw. $x = l$ gilt, durch Anzahl aller Möglichkeiten, d.h. durch die Größe der Menge S:

$$\begin{array}{rcl}
Prob(min\{x,y\} = 1) & = & \dfrac{2((i-1)-1)}{(i-1)(i-2)} \\
Prob(min\{x,y\} = 2) & = & \dfrac{2((i-1)-2)}{(i-1)(i-2)} \\
\vdots & \vdots & \vdots \\
Prob(min\{x,y\} = k-1) & = & \dfrac{2((i-1)-(k-1))}{(i-1)(i-2)} \\
Prob(min\{x,y\} = k+1) & = & \dfrac{2((i-(k+1))}{(i-1)(i-2)} \\
\vdots & \vdots & \vdots \\
Prob(min\{x,y\} = i-1) & = & \dfrac{2}{(i-1)(i-2)}.
\end{array}$$

Für $l < k$ ist damit ist $Prob(B_l \mid A_k) = \frac{2((i-1)-l)}{(i-1)(i-2)}$ und

$$q(l,k) = \frac{1}{i}\frac{2(i-l-1)}{(i-1)(i-2)}. \tag{61}$$

Dieses Ergebnis bezieht sich auf einen *durchschnittlichen* Weak-Heap der Größe i. Darum läßt sich die Anzahl der zulässig zu generierenden Weak-Heaps der Größe $i+1$, die durch $MergeForestUp(j)$ gebildet werden, wie folgt berechnen:

$$\begin{array}{rcl}
F(i,j) & = & i!\displaystyle\sum_{k=1}^{i}\sum_{l=1}^{k-1} q(l,k) \\
& = & i!\displaystyle\sum_{l=1}^{i-1}\sum_{k=l+1}^{i} q(l,k) \\
& = & 2(i-3)!\displaystyle\sum_{l=1}^{i-1}\sum_{k=l+1}^{i} (i-l-1) \\
& = & 2(i-3)!\displaystyle\sum_{l=1}^{i-1}(i-l)((i-l)-1) \\
& = & 2(i-3)!\left(\displaystyle\sum_{l=1}^{i-1}(i-l)^2 - \sum_{l=1}^{i-1} i-l\right) \\
& = & 2(i-3)!\left(\displaystyle\sum_{l=1}^{i-1} l^2 - \sum_{l=1}^{i-1} l\right)
\end{array}$$

$$
\begin{aligned}
&= 2(i-3)!\left(\frac{i(i-1)(2i-1)}{6} - \frac{i(i-1)}{2}\right) \\
&= 2(i-3)!\left(\frac{i(i-1)(2i-1)}{6} - \frac{i(i-1)}{2}\right) \\
&= \frac{i!}{i-2}\left(\frac{(2i-1)}{3} - \frac{3}{3}\right) \\
&= \frac{i!}{i-2}\frac{(2i-4)}{3} \\
&= \frac{2i!}{3}.
\end{aligned}
$$

Mit

$$f(i,j) = \frac{F(i,j)}{\frac{(i+1)!}{2}} \tag{62}$$

ergibt sich die Behauptung. □

Definition 9 *Der Wahrscheinlichkeitsraum S wird über folgendes stochastisches Experiment definiert: In Urne U_i, $i \in \{0,\ldots,n-1\}$ liegen $(i+1)!/2$ Kugeln entsprechend der Anzahl zulässig zu generierender Weak-Heaps. Jede Kugel ist mit dem Index j beschriftet, der zur Generierung mittels $MergeForestUp$ gewählt wurde. Im Schritt i wird aus der Urne U_{i-1} eine Kugel gezogen und deren Beschriftung s_i betrachtet. Damit ist*

$$S = \{(s_1,\ldots,s_n) \mid s_i \in \{0,\ldots,i-1\}\}. \tag{63}$$

Definition 10 *Die Indikatorzufallsvariablen $Z_i : S \to \{0,1\}$ seien wie folgt definiert:*

$$Z_i = \begin{cases} 0 & \textit{falls } s_i \in R_i \cap Leaf(i), \\ 1 & \textit{sonst.} \end{cases}$$

Dabei bezeichne $Leaf(i)$ die Menge aller Blätter in einem Heap der Größe i.

Die Zufallsvariablen beschreiben jeweils ein Bernoulli-Experiment mit dem Erfolgsparameter

$$p_i = Prob(Z_i = 1) = 1 - Prob(Z_i = 0) = 1 - \sum_{j \in R_i \cap Leaf(i)} f(i,j).$$

Demnach ist der Erwartungswert $E[Z_i] = p_i$. Betrachte erneut die Rückwärtsanalyse: Ist $Z_i = 0$, so kann ein Vergleich zum worst-case eingespart werden.

Somit gibt der Wert von $\sum_{i=1}^n (1 - Z_i)$ ein Maß für die Mindesteinsparung an Vergleichen. Im Mittel sind dies

$$\begin{aligned}
E[\sum_{i=1}^n (1 - Z_i)] &= \sum_{i=1}^n E[1 - Z_i] \\
&= \sum_{i=1}^n \left(1 - \left(1 - \sum_{j \in R_i \cap Leaf(i)} f(i,j)\right)\right) \\
&\overset{HS42}{=} \sum_{i=1}^n \mid R_i \cap Leaf(i) \mid \cdot \frac{4}{3(i+1)} \\
&= \sum_{i=1}^n (2^{\lfloor \log(i-1) \rfloor} - 1 - \lfloor \frac{i-1}{2} \rfloor) \cdot \frac{4}{3(i+1)}
\end{aligned}$$

viele.
Ziel der folgenden Untersuchung wird es sein, diese Summe als $(4/3 \ln 2 - 2/3)n - O(\log n)$ darzustellen.

Hilfssatz 43 *Sei H_n die n–te Partialsumme der harmonischen Reihe, d.h. $H_n = \sum_{i=1}^n \frac{1}{i}$. Dann existieren Konstanten c, c', so daß für alle n:*

$$\ln n + \gamma - c/n \leq H_n \leq \ln n + \gamma + c'/n,$$

wobei γ die Eulersche Konstante ist (vergl. Anhang).

Beweis: (Graham (1990)). □

Hilfssatz 44 *Es gilt:*

$$\ln(1 + 1/n) = O(1/n). \tag{64}$$

Beweis: Taylorexpansion. □

Hilfssatz 45

$$\sum_{i=0}^{n-1} \frac{2^{\lfloor \log i \rfloor}}{i+2} \geq \ln 2 \cdot n - O(\log n). \tag{65}$$

Beweis: Die einzelnen Summanden lassen sich übersichtlich in einem binären Baum beschreiben (vergl. Abb. 27). Ziel wird es sein, die Summanden levelweise durch Teilstücke der Harmonischen Reihe zu beschreiben.
Es existieren Konstanten c, c', so daß für den m-ten Level gilt:

$$2^m (H_{2^{m+1}+1} - H_{2^m+1}) \quad \geq \quad 2^m \left\{ \ln(2^{m+1} + 1) + \gamma - \frac{c}{2^{m+1} + 1} \right.$$

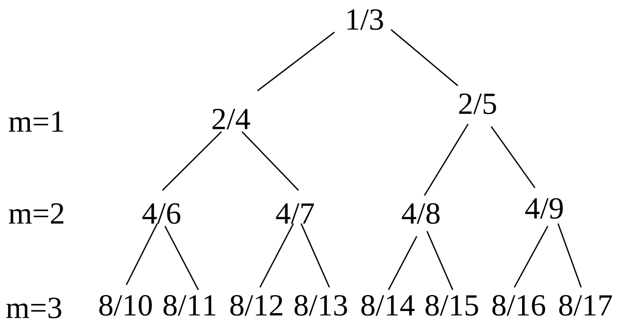

Abbildung 27: Ein Baum, der levelweise eine Summe beschreibt.

$$
\begin{aligned}
& \left. - \ln(2^m + 1) - \gamma - \frac{c'}{2^m+1} \right\} \\
\geq \quad & 2^m \left\{ \ln \frac{2^{m+1}+1}{2^m+1} - \frac{c+c'}{2^m} \right\} \\
= \quad & 2^m \left\{ \ln \left(2 - \frac{1}{2^m+1} \right) - \frac{c+c'}{2^m} \right\} \\
= \quad & 2^m \left\{ \ln 2 + \ln \left(1 - \frac{1}{2^m+1} \right) - \frac{c+c'}{2^m} \right\} \\
= \quad & 2^m \left\{ \ln 2 - \frac{c+c'}{2^m} \right\} \\
= \quad & 2^m \ln 2 - (c+c').
\end{aligned}
$$

Damit gilt für die Konstante $c^* = c + c'$:

$$
\begin{aligned}
\sum_{i=0}^{n-1} \frac{2^{\lfloor \log i \rfloor}}{i+2} \quad & \geq \quad \frac{1}{3} + \sum_{m=1}^{k-1} 2^m (H_{2^{m+1}+1} - H_{2^m+1}) \\
& \geq \quad \frac{1}{3} + \sum_{m=1}^{k-1} (2^m \ln 2 - c^*) \\
& = \quad \frac{1}{3} + \ln 2 \sum_{m=1}^{k-1} 2^m - \sum_{m=1}^{k-1} c^* \\
& = \quad \frac{1}{3} + \ln 2 (2^k - 2) - O(k) \\
& = \quad \ln 2 \cdot n - O(\log n). \square
\end{aligned}
$$

Hilfssatz 46

$$\sum_{i=1}^{n} \frac{\lfloor \frac{i+1}{2} \rfloor}{\frac{i+1}{2}} = n - O(\log n). \tag{66}$$

Beweis: Fallunterscheidung: Ist i gerade so ist:

$$\frac{\lfloor \frac{i}{2} \rfloor}{\frac{i}{2}} = 1.$$

Ist i ungerade, so gilt:

$$\frac{\lfloor \frac{i}{2} \rfloor}{\frac{i}{2}} = \frac{\frac{i-1}{2}}{\frac{i}{2}} = \frac{i-1}{i} = 1 - \frac{1}{i}.$$

Damit existieren $(n-1)+1 = n$ Summanden 1 und für den Minuend ergibt sich:

$$\sum_{i=1}^{\lfloor n/2 \rfloor} \frac{1}{2i+1} < H_n \in O(\log n),$$

wobei H_n die n-te Partialsumme der harmonischen Reihe beschreibt.□

Satz 41 *Die mittlere Anzahl von Schlüsselvergleichen von WEAK-HEAPSORT ist unter der Modellannahme* (M) *durch* $n \log n - 0.1715n + O(\log n)$ *nach oben beschränkt.*

Beweis: Die mittlere Anzahl von Einsparungen an Vergleichen ist:

$$\begin{aligned}
E[\sum_{i=1}^{n}(1 - Z_i)] &= \sum_{i=1}^{n} \left(2^{\lfloor \log(i-1) \rfloor} - 1 - \lfloor \frac{i-1}{2} \rfloor \right) \cdot \frac{4}{3(i+1)} \\
&= \frac{4}{3} \sum_{i=1}^{n} \frac{2^{\lfloor \log(i-1) \rfloor}}{i+1} - \frac{2}{3} \sum_{i=1}^{n} \frac{\lfloor \frac{i+1}{2} \rfloor}{\frac{i+1}{2}} \\
&\geq \frac{4}{3} (\ln 2 \cdot n - O(\log n)) - \frac{2}{3} (n - O(\log n)) \\
&= \left(\frac{4}{3} \ln 2 - \frac{2}{3} \right) n - O(\log n) \\
&\approx 0.2575n - O(\log n).
\end{aligned}$$

Die maximale Anzahl von Schlüsselvergleichen ist durch $n \log n + 0.086013n$ nach oben beschränkt. □
Die nur unter Modellannahmen bewiesene Grenze für den average-case von WEAK–HEAPSORT wird in Experimenten noch weit unterboten:

Vermutung 2 *Sei* $d(n)$ *so gewählt, daß* $n \log n + d(n)$ *die erwartete Anzahl von Vergleichen für WEAK–HEAPSORT ist. Dann ist* $d(n) \in [-0.47, -0.42]$*. Weiterhin ist* $d(n)$ *klein für* $n \approx 2^k$ *und groß für* $n \approx 1.4 \cdot 2^k$*.*

Beleg: Die Vermutung läßt sich durch Experimente bekräftigen. Dabei werden die Ergebnisse in Tabelle 5 so dargestellt, daß sie sich gut mit denen von BOTTOM–UP–HEAPSORT (Wegener (1993)) vergleichen lassen:

n	1000	2000	3000	4000	5000	6000	7000	8000
$d_{exp}(n)$	-0.462	-0.456	-0.437	-0.456	-0.445	-0.429	-0.436	-0.458
n	9000	10000	11000	12000	13000	14000	15000	16000
$d_{exp}(n)$	-0.448	-0.437	-0.432	-0.430	-0.436	-0.443	-0.449	-0.458
n	17000	18000	19000	20000	21000	22000	23000	24000
$d_{exp}(n)$	-0.458	-0.449	-0.443	-0.437	-0.433	-0.431	-0.436	-0.427
n	25000	26000	27000	28000	29000	30000		
$d_{exp}(n)$	-0.431	-0.437	-0.436	-0.440	-0.440	-0.447		

Tabelle 5: Empirische Bestimmung des average-cases von WEAK–HEAPSORT.

Hierbei war es nahezu unerheblich, ob ein Experiment durchgeführt worden ist, oder über 20 Versuche gemittelt wurde. Dies begründet sich dadurch, daß die Varianz der Vergleichsanzahl von WEAK-HEAPSORT sehr klein ist: Für n=30000 befand sich der best-case von 20 durchgeführten Versuchen bei 432657 und der worst-case bei 432816.
WEAK-HEAPSORT benötigt somit im Mittel ca. $0.81n$ weniger Schlüsselvergleiche als BOTTOM-UP-HEAPSORT (vergl. Vermutung 1) und ca. $0.45n$ weniger als MDR-HEAPSORT (vergl. Wegener (1992)).

11.3 Rückwärtige Generierung aller Weak-Heaps

Ich bin ein Teil von allen, denen ich begegnet bin.

Alfred Lord Tennyson

Die Ergebnisse der Rückwärtsanalyse ermöglichen, den Konfigurationenbaum rückwärtig aufzubauen.
Die Prozedur S berechnet rekursiv alle möglichen Paarfolgen $(z[i], r[i])$, $i \in \{0, \ldots, n-1\}$, die mögliche Eingaben für die im Kapitel 8 beschriebene $MergeForestUp$–Prozedur bzw. die anschließende Belegung der Reversebits darstellen.
Genauer betrachtet bezeichnet $z[i]$ nicht den Index des Elementes, das mittels $MergeForestUp$ an die Wurzel getragen wird, sondern den Schlüssel an der betreffenden Stelle. Die Veränderungen von $MergeForestUp$ sind jedoch einfach

durch das Ändern des formalen Parameters und der Zuhilfenahme einer Suchprozedur *Find* zu verwirklichen. Diese Prozedur ermittelt, ob ein Element sich in einem angegebenen Teilbaum enthalten ist. Dies führt zur folgenden Prozedur $MergeForestUp^*$:

```
PROCEDURE MergeForestUp*(a[j])
  x = 1
  WHILE (a[j] <> a[0]) DO
    IF Find(a[j],rT(x)) OR (a[j] = a[x]) THEN
       SWAP(a[0],a[x])
       REVERSE[x] = 1 - Reverse[x]
    FI
    x = 2x + Reverse[x];
  OD
END MergeForestUp*.
```

Ist zum Beispiel $n = 4$, so ist eine der durch S gebildetet Paarfolgen $((0,1),(1,1),(0,2),(1,4))$. Es gibt insgesamt $n!2^n$ viele Paarfolgen. Die Prozedur S gestaltet sich nun wie folgt:

```
PROCEDURE S(i)
  IF i = 0 THEN
    a[0] = 1                          { Ein Weak-Heap }
    Reverse[0] = 0                    { der Groesse 1 }
    FOR j = 1 TO n DO a[j] = 0        { wird initialisiert   }
    FOR j = 1 TO n DO Reverse[j] = 0
    FLAG = TRUE                       { * }
    l = 1                             { Schleifenvariable }
    WHILE (l <= n) AND FLAG DO
      FLAG = Generate(l,r[l-1],z[l-1])
      Inc(l)
    OD
  ELSE
    FOR j = 1 TO n-i+1 DO BEGIN
      z[n-i] = j
      r[n-i] = 0
      S(i-1)
      r[n-i] = 1
      S(i-1)
  END
END S.
```

Die aufgerufene Prozedur $Generate(l, z[l-1], r[l-1])$, $l \in \{1, \ldots, n\}$, prüft, ob die Eingabe legal ist, d.h. ob der übergebene z–Wert nicht größer als $s_l = a[Gprarent(l)]$ ist. Dies ist gleichzeitig der Rückgabewert der Funktion, um einen rechtzeitigen Abbruch zu ermöglichen. Somit werden von den $n!2^n$ vielen potentiellen Weak-Heaps die $n!$ zulässigen ermittelt. Desweiteren werden die additiven Vertauschungen und Reversebitbelegungen vorgenommen, um den Rückwärtsschritt zu $MergeForestUp^*(k = z[l-1])$ zu komplettieren.

```
PROCEDURE Generate(l,k,Rev):BOOLEAN
  IF k <= a[Gparent(l)] THEN
    Reverse[l] = Rev                     { ** }
    a[l] = l+1
    MergeForestUp*(k)
    Swap(a[0],a[l])
    Generate = TRUE
  ELSE
    Generate = FALSE
  FI
END Generate.
```

Nun ist es von Interesse, bei einer vorgegebenen Kette von Fällen (I) und (II), die Anzahl der unter dieser Vorgabe erzeugbaren Weak-Heaps mit einer Zählvariable *count* zu ermitteln. Dazu sei Fall(I) mit FALSE und Fall(II) mit TRUE gekennzeichnet. Weiterhin sei $Kette$ ein Array Boole'scher Variablen, mit der Eigenschaft, daß $Kette[i]$, $i \in \{1, \ldots, n\}$, den vorliegenden Fall im i-ten Schritt der Rückwärtsanalyse beschreibt. Es kann nur im n-ten Schritt gezählt werden, wenn sowohl alle Weak-Heaps auf dem Weg zulässig waren, als auch die entsprechenden Fälle aufgetreten sind. Die erste Bedingung wird mit Hilfe von einem Array Boole'scher Variablen $link$ realisiert, welches initial FALSE (Stelle (*)) ist und beim Erreichen letzten Schrittes TRUE wird. Die Prozedur $Check(l)$ übernimmt den Prüf- und Zählvorgang und wird an der Stelle (**) in den Quellcode eingefügt:

```
PROCEDURE Check(l)
  FOR k = 1 TO n-1 DO
    IF (l = k) AND (Fall(l,Position(k))=Kette[l])
               AND (link[k-1]=TRUE) THEN link[k] = TRUE
  OD
  IF (l = n) AND (Fall(l,Position(k))=Kette[l])
             AND (link[n-1]=TRUE) THEN inc(count)
END Check.
```

Dabei ermittelt $Position(k)$ den Index des Wertes k und $Fall(l, Position(k))$ den auftretenden Fall. Diese Prozedur wurde im vorangegangenen Abschnitt beschrieben.

Es lassen sich somit die Verteilungen der Weak-Heaps für feste Fallfolgen ermitteln. Faßt man diese Ergebnisse nach der Häufigkeit der auftretenden Fälle zusammen, so ergibt sich die gleiche Verteilung, die sich auch durch die Vorwärtsbetrachtung der Vergleichsanzahlen ergibt.

Beispiel 14 *Sei $n = 6$. Es gibt 144 Weak-Heaps, in denen jedesmal der ungünstige Fall(II) vorliegt. Desgleichen gibt es 408 Weak-Heaps, bei denen der günstige Fall(I) genau einmal vorliegt und 168 Weak-Heaps wo selbiger zweimal vorliegt.*

Dieses Verhalten war zu erwarten, belegt dennoch die Korrektheit des gewählten Ansatzes.

12 Eine Datenstruktur für diverse Operationen

Die Erfindung des Problems ist wichtiger als die Erfindung der Lösung.

Walther Rathenau

12.1 Das MIN-MAX Problem

Ich jage nie zwei Hasen auf einmal.

Otto vonBismark

Die Bestimmung des maximalen Elementes von n Objekten benötigt notwendigerweise $n-1$ Vergleiche, da kein zusammenhängender Graph auf n Punkten mit $n-2$ Kanten existiert. Falls eine Kante (u,v) jeweils einem Vergleich zweier Objekte u und v entspricht, so kann nicht entschieden werden, in welcher Zusammenhangskomponente das maximale Element liegt. Die Generierungsphase des WEAK-HEAPSORT-Algorithmus liefert das maximale Element demnach in optimalen $n-1$ Schritten. Dabei fällt die gesamte Weak-Heap-Datenstruktur nahezu als *Abfallprodukt* ab.
Der Zugewinn an Struktur kann auch in der optimalen Lösung eines weiteren Problemes erläutert werden.

Definition 11 *Seien $a_1, \ldots, a_n$ paarweise verschiedene Werte. Das MIN-MAX Problem fragt nach der gleichzeitigen Berechnung des Minimums und des Maximums dieser Werte.*

Satz 42 *Es sind mindestens $n + \lceil n/2 \rceil - 2$ Vergleiche notwendig, um das MIN-MAX-Problem zu lösen.*

Beweis: (Wegener (1991)[15]) Sei ein korrektes Verfahren V zur Lösung des MIN-MAX-Problems gegeben. Seien S_{MIN} (bzw. S_{MAX}) die Mengen derjenigen Elemente, die in einem Zustand von V sicher größer (kleiner) sind als das Minimum

[15] Ebd. wird der Beweis in der *Sprache der Tuniere geführt*

(Maximum) aller. Initial ist $S = \mid S_{MIN} \mid + \mid S_{MAX} \mid = 0$ und zum Ende von V ist $S = 2n - 2$. Bezeichne mit

$$comp(a,b) = (min\{a,b\}, max\{a,b\}) = \left(\frac{a+b}{2} - \left|\frac{a-b}{2}\right|, \frac{a+b}{2} + \left|\frac{a-b}{2}\right|\right)$$

die Vergleichsoperation zweier Werte a, b. Eine Anwendung von $comp(a,b)$ kann die Mengen S_{MIN} und S_{MAX} vergrößern. Für S ergibt sich ein Summand von

- 0 (ungünstig) oder 2 (günstig), falls $(a,b) \in S_{MIN} \times S_{MAX} \cup S_{MAX} \times S_{MIN}$,
- 2, falls $a, b \notin S_{MIN} \cup S_{MAX}$,
- 1, falls $(a,b) \in S_{MIN} \times S_{MIN} \cup S_{MAX} \times S_{MAX}$,
- 1 (ungünstig) oder 2 (günstig), sonst.

Der zweite Fall kann maximal $\lfloor n/2 \rfloor$ mal auftreten. Daher ist die Anzahl der Vergleiche im ungünstigen Fall mindestens:

$$2\lfloor n/2 \rfloor + 1\,(2n - 2 - \lfloor n/2 \rfloor) = n + \lceil n/2 \rceil - 2. \square \qquad (67)$$

Mit der Hilfe von in $n-1$-Vergleichen generierten Max-Weak-Heaps läßt sich diese untere Schranke leicht erreichen. Das Maximum liegt an der Wurzel an. In weiteren $\lceil n/2 \rceil - 1$ Vergleichen kann das sich auf dem letzten Level befindliche Minimum gefunden werden:
Es liegt für gerades n an den Positionen $\lfloor (n-1)/2 \rfloor + 1$ bis n und für ungerades n an den Positionen $\lfloor (n-1)/2 \rfloor$ bis n, d.h in beiden Fällen an den Positionen $\lfloor n/2 \rfloor$ bis n. Dieses sind $n - 1 - \lfloor n/2 \rfloor + 1 = \lceil n/2 \rceil$ viele Positionen. Somit werden $\lceil n/2 \rceil - 1$ Vergleiche benötigt.
Mit exakt der gleichen somit optimalen Vergleichsanzahl zusätzlich ein Min-Weak-Heap gebildet werden:

Satz 43 *In $n + \lceil n/2 \rceil - 2$ Vergleichen kann ein Max-Weak-Heap und ein Min-Weak-Heap gleichzeitig generiert werden.*

Beweis: (Dutton (1992)) Sei ein Max-Weak-Heap in $n-1$ Schritten erzeugt. Die Idee ist, die an den Blättern i, $i \in \{\lfloor (n-1)/2 \rfloor + 1, \ldots, n\}$, anliegenden Schlüssel im Aufbau des Min-Weak-Heaps nicht mit $Gparent(i)$ zu vergleichen, sondern nur zu vertauschen, da gesichert ist, daß deren Wert größer ist. Von dem Index $\lfloor (n-1)/2 \rfloor$ ausgehend wird der Min-Weak-Heap analog zum Max-Weak-Heap unter der Änderung der Ungleichung in der $Merge$-Prozedur aufgebaut.

```
PROCEDURE MaxToMin
  FOR i = n-1 DOWNTO (n-1) div 2 + 1 DO
    Swap(Gparent(i),i)
```

```
  OD
  FOR i = (n-1) DIV 2 DOWNTO 1 DO
    MinMerge(Gparent(i),i)
  OD
END MaxToMin.
```

Da $\lfloor (n-1)/2 \rfloor = \lceil n/2 \rceil - 1$ benötigt der Algorithmus die optimale Anzahl von $n + \lceil n/2 \rceil - 2$ Vergleichen, um einen Min-Weak-Heap als auch einen Max-Weak-Heap zu generieren. □

12.2 Konvertierung eines Heaps

Nur der Wechsel ist wohltätig. Unaufhörliches Tageslicht ermüdet.

Wilhelm von Humboldt

Weak-Heaps wurden eingeführt, da sich die Heap-Bedingung als zu stark erwies, um in der Auswahlphase schnell genug wieder hergestellt werden zu können. Die schwächere Struktur Weak-Heaps läßt sich auch dadurch erkennen, daß sich ein Heap ohne zusätzliche Vergleiche direkt In-Situ in einen Weak-Heap umwandeln läßt.

Definition 12 $\overline{S_x} = \{$ *rchild(x)* $\} \cup \{y \mid y$ *ist von rchild(x) nur über rechte Kinder erreichbar* $\}$.

Satz 44 *Sei* $a[1], \ldots, a[n]$ *ein (eingebetteter) Heap auf den Elementen aus einer geordneten Menge* S*. Sei* $k = \lceil \log n \rceil$ *und* $n = (b_k \ldots b_0)_2$*. Die Zuweisungen:*

1. $a[0] = a[1]$,
2. $Reverse[i] = 0$, $i \in \{0, \ldots, n-1\}$,
3. $a[\lfloor \frac{n}{2^{k-i+1}} \rfloor] = a[\lfloor \frac{n}{2^{k-i}} \rfloor]$, $i \in \{2, \ldots, k\}$,
4. $Reverse[\lfloor \frac{n}{2^{k-i+1}} \rfloor] = 1 - \left(\lfloor \frac{n}{2^{k-i}} \rfloor \bmod 2\right) = 1 - b_{k-i}$, $i \in \{1, \ldots, k\}$

transformieren den Heap in einen Weak-Heap der Größe n*.*

Beweis: Es sind $(W1)$ bis $(W3)$ der Definition eines Weak-Heaps zu zeigen.

zu (W1) Da $Reverse[0] = 0$ ist, hat die Wurzel keinen linken Teilbaum. An der Stelle 0 steht das maximale Element.

zu (W2) Der spezielle Pfad $SP_H = \{\lfloor \frac{n}{2^i} \rfloor \mid i \in \{1, \ldots, k\}\}$ beschreibt im Heap den Weg von dem Index n zurück zum Index 1. Weiterhin gilt ($\div$ steht für mod, r für *Reverse*):

$$
\begin{array}{lclclclcl}
n \div 2 & = & b_0 & \Rightarrow & n & = & 2\lfloor \frac{n}{2^1} \rfloor + b_0 & = & 2\lfloor \frac{n}{2^1} \rfloor + 1 - r[2\lfloor \frac{n}{2^1} \rfloor] \\
\lfloor \frac{n}{2^1} \rfloor \div 2 & = & b_1 & \Rightarrow & \lfloor \frac{n}{2^1} \rfloor & = & 2\lfloor \frac{n}{2^2} \rfloor + b_1 & = & 2\lfloor \frac{n}{2^2} \rfloor + 1 - r[2\lfloor \frac{n}{2^2} \rfloor] \\
\ldots & = & \ldots & \Rightarrow & \ldots & = & \ldots & = & \ldots \\
\lfloor \frac{n}{2^{k-2}} \rfloor \div 2 & = & b_k & \Rightarrow & \lfloor \frac{n}{2^{k-2}} \rfloor & = & 2\lfloor \frac{n}{2^{k-1}} \rfloor + b_{k-2} & = & 2 \cdot 1 + 1 - r[1].
\end{array}
$$

Damit wird jedoch der Pfad $\overline{S_x}$ (rückwärts gelesen) beschrieben, wobei beachtet werden muß, daß nicht der Index n sondern $\lfloor \frac{n}{2} \rfloor$ sich als dessen letztes Element ergibt.

Sei $z_i = \lfloor \frac{n}{2^{k-i}} \rfloor$, $i \in \{2, \ldots, k\}$. Für den Wert b_{k-i} ergeben sich zwei Fälle:

b_{k-i}**=1:** Es ist z_i in dem Heap rechtes Kind von $\lfloor (z_i/2) \rfloor$ und es ergibt sich ein in Abb. 28 dargestelltes Bild.

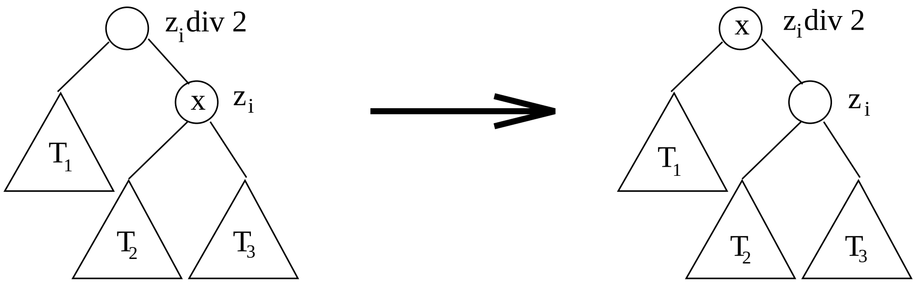

Abbildung 28: Erster Fall bei der Umformung eines Heaps in einen Weak-Heap.

Im Heap galt für alle $y \in T_2 \cup T_3$, daß $x \geq a[y]$ ist, somit gilt in dem Weak-Heap für alle $y \in rT(\lfloor (z_i/2) \rfloor) : a[\lfloor (z_i/2) \rfloor] \geq a[y]$.

b_{k-i}**=0:** Es ist z_i in dem Heap linkes Kind von $\lfloor (z_i/2) \rfloor$ und es ergibt sich ein in Abb. 29 dargestelltes Bild.

Da das Reversebit an der Stelle $\lfloor (z_i/2) \rfloor$ gesetzt wird, überträgt sich die Argumentation von dem Fall $b_{k-i} = 1$.

Für Elemente $x \notin SP_H$ gilt für alle $y \in rT(x) : a[x] \geq a[y]$, da die Transformation keine Veränderung des durch x beschriebenen Teilbaumes bewirkt.

zu (W3) Der Weg SP_H endet an einem Blatt maximaler Tiefe. Damit bleibt die Tiefendifferenz auch nach der Ungültigkeit vom Index n für Knoten mit weniger als 2 Kindern exklusive der Wurzel maximal 1. Weder die Vertauschung von Werten noch die Belegung der Reversebits zerstören diese Invarianz. □

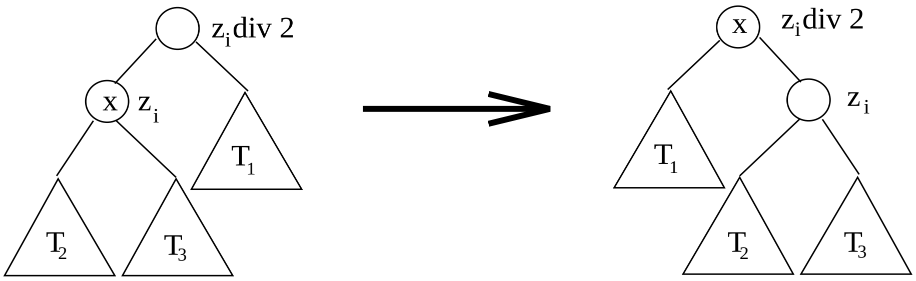

Abbildung 29: Zweiter Fall bei der Umformung eines Heaps in einen Weak-Heap.

Beispiel 15 *Sei $n = 10$, d.h. $n = (1\ 0\ 1\ 0)_2$. Die im Satz 44 beschriebene Transformation wird durch folgende Abbildung veranschaulicht:*

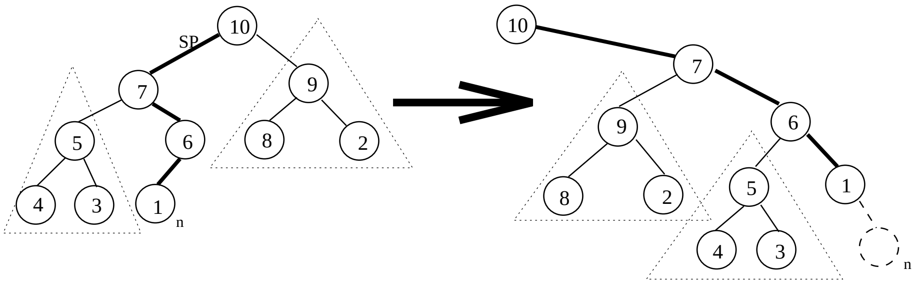

Abbildung 30: Beispiel der Umformung eines Heaps in einen Weak Heap.

12.3 Der Weak-Heap als Priority Queue

Das Gute ist zweimal so gut, wenn es kurz ist.

Baltasar Grácian
Handorakel der Weltklugheit

Priority Queues tauchen als Datenstruktur in der Lösung von vielerlei Problemen der Infomatik auf, z.B. im Algorithmus von Dijkstra zur Berechnung der kürzesten Wege in einem kantenbewerteten Graphen.

Definition 13 *Sei S eine geordnete Menge. Ein Datenstruktur, die das Einfügen eines Elementes $v \in S$ $(Insert(v))$ und das Entfernen des maximalen Wertes $(DeleteMax)$ ermöglicht, heißt Priority Queue.*

Heaps eignen sich gut für den Einsatz als Priority Queue, da beide Operationen in logarithmisch vielen Schritten durchführbar sind.

Es soll untersucht werden, ob sich $Insert$ und $DeleteMax$ mit Hilfe von Weak-Heaps auch effizient realisieren lassen.
Die Idee des Einfügens (Dutton (1992)[16]) besteht darin, von einer freien Stelle auf dem untersten Level (hier: Index n) solange $Merge(Gparent(i), i)$ aufzurufen, bis die Bedingung in der $Merge$-Prozedur erfüllt oder die Wurzel erreicht ist.

```
PROCEDURE Insert(v)
  x = n
  WHILE x <> 0 DO
    IF a[Gparent(x)] < a[x] THEN
      Swap(a[Gparent(x)],a[x])
      Reverse[x] = 1 - Reverse[x]
      x = Gparent(x)
    ELSE
    FI
  OD
END Insert.
```

Hilfssatz 33 belegt, daß ein Abbruch an der Stelle, wo $a[Gparent(x)] \geq a[x]$ gilt, die Bedingung $(W1)$ sichert. Bis zu diesem Zeitpunkt werden nur $Merge(Gparent(x), x)$-Operationen durchgeführt. Somit sichert Satz 2 wie schon in der Generierungsphase, daß die so entstehenden Datenstruktur ein Weak-Heap ist.
Es soll die Länge des *großelterlichen* Weges vom Index n zur Wurzel als obere Grenze für die Anzahl an Operationen der Insert analysiert werden:

Definition 14 *Der großelterliche Pfad wird rekursiv wie folgt festgelegt:*

$$GEP_x = \{x\} \cup GEP_{Gparent(x)}.$$

Je nach Belegung der Reversebits kann n an einer unbestimmten Position auf dem letzten Level liegen.

Hilfssatz 47 *Die maximale Länge des Weges GEP_n ist $\lceil \log(n+1) \rceil$.*

Beweis: Ein Weak-Heap mit $n+1$ Elementen besitzt den größten Index n. Demnach ist die Tiefe gleich $\lceil \log(n+1) \rceil$. Falls immer $Gparent(x) = Parent(x)$ gilt, sprich wenn x immer rechtes Kind von $Parent(x)$ ist, dann wird somit $\lceil \log(n+1) \rceil$ mal auf $Gparent(x)$ verwiesen, bis der Index 0 erreicht ist. □
Damit ist das Einfügen in einen Weak-Heap in $O(\log n)$ Schritten gewährleistet.

Hilfssatz 48 *Sei $n = 2^k$ und für alle $i \in \{0, \ldots, n-1\}$ sei $Prob(Reverse[i] = 0) = 1/2$. Die durchschnittliche Länge von GEP_n beträgt $k/2 + 1$.*

[16] Hier muß (!) *Gparent* mit dem Prädikat `odd(j) = Reverse(j DIV 2)` implementiert sein.

Beweis: Da die Belegung der Reversebits unabhängig vom gewählten Index ist, sind die Positionen auf dem letzten Level für n gleichwahrscheinlich. Die Summe der Längen ($SL(n)$) von den Pfaden GEP_n über alle möglichen Positionen für n genügt folgender Rekursionsgleichung (vergl. Abb 7):

$$SL(1) = 1, \quad (68)$$

$$SL(n) = 2SL(n/2) + n/2. \quad (69)$$

Durch Induktion verifiziert man leicht die geschlossene Form $SL(n) = nk/2+n$. Damit beträgt die erwartete Länge von $GEP_n = SL(n)/n = k/2+1$. □
Die Prozedur $DeleteMax$ entspricht einem Schritt der Auswahlphase des WEAK-HEAPSORT-Algorithmus.

```
PROCEDURE DeleteMax
  x = a[0]
  MergeForest*(n)
  return(x)
END DeleteMax.
```

Der Beleg über die Korrektheit dieses Vorgehens überträgt sich somit. Die Betrachtungen über die Verbesserungen des Algorithmus legten offen, daß im Falle a (vergl. Abb 8) ein Vergleich eingespart werden kann:
Falls das letzte Element des speziellen Weges den Index m ungleich n hat, so wird es an dessen Stelle als Vergleichselement für die $Merge$–Prozedur verwendet. Somit sind für jedes $DeleteMax$ exakt $\lceil \log(n+1) \rceil - 1$ wesentliche Vergleiche notwendig.

13 Die Sortieralgorithmen in der Praxis

Einmal selbst sehen ist mehr wert als hundert Neuigkeiten hören.

Japanisches Sprichwort

Die Schlüsselvergleichsanzahl ist ein, wenn nicht sogar das ausschlaggebende Kriterium für das Laufzeitverhalten eines allgemeinen Sortierverfahrens. Die Tauschoperationen von Schlüsseln lassen sich, wie schon im Vorwort beschrieben, durch den Tausch der auf die entsprechenden Schlüssel verweisenden Zeiger simulieren. Dadurch wird zusätzlicher Speicher linearer Größe verbraucht, so daß häufig auf diese Option verzichtet wird. Damit erscheint in diesem Fall eine gleichberechtigte Bewertung von Vertauschungen und Vergleichen fairer.
Weiterhin wurde erarbeitet, daß sich SHELLSORT, QUICKSORT, CLEVER--QUICKSORT, BOTTOM-UP– bzw. MDR-HEAPSORT und WEAK-HEAPSORT in der Anzahl der benötigten Vergleiche im Mittel nicht allzu stark unterscheiden, so daß es fraglich bleibt, inwieweit sich unwesentliche Operationen

(Zuweisungen von Zeigern, rekursive Funktionsaufrufe, Vergleiche und Zuweisungen von Indizes, etc.) auf das Laufzeitverhalten der Algorithmen auswirken. Um diesen Beobachtungen und Fragen Rechnung zu tragen, werden zwei Wege beschritten.
Auf der einen Seite werden die Algorithmen einem Wettkampf gleich gemäß gegebener Gewichtung für Schlüsselvergleiche, Schlüsselvertauschungen und Funktionsaufrufen *aktiv* miteinander verglichen.

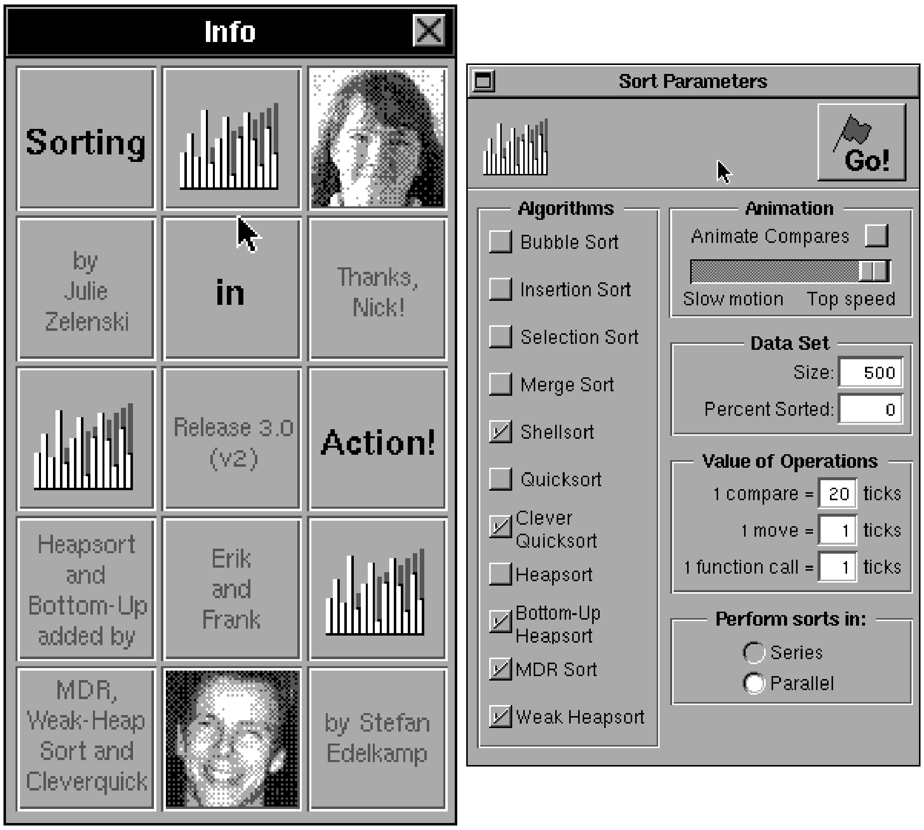

Abbildung 31: Das Info–Fenster und das Fenster für die Parameterisierung des Programms *Sorting in Action*.

Dazu wurde das Programm *Sorting in Action*, das von von Julie Zelenski für das NEXTSTEP-Betriebssystem geschrieben wurde, um die entsprechenden, oben angegebenen Algorithmen erweitert (vergl. Abb. 31).
Der Sortiervorgang kann nach Festlegung der Gewichte (sogenannten Ticks), der Eingabegröße n, den Grad der Vorsortiertheit und der Ausführungsgeschwindigkeit am Bildschirm verfolgt werden. Dabei werden die Verfahren bezüglich der

verbrauchten Ticks synchronisiert.
Es werden SHELLSORT, CLEVER-QUICKSORT, BOTTOM-UP- und MDR-HEAPSORT zur Herausforderung von WEAK-HEAPSORT herangezogen.
Abbildung 32 zeigt, von rechts oben nach links unten betrachtet, die Bildschirminhalte zum Beginn des Sortiervorganges, nach der Aufbauphase von WEAK-HEAPSORT, ca. in der Mitte der Auswahlphase und zum Ende des Sortiervorganges. Dabei ist ein Vergleich mit 20 Ticks, eine Zuweisung und ein rekursiver Funktionsaufruf jeweils mit einem Tick bewertet.
Zu der Implementierung der konkurierenden Algorithmen ist folgendes anzumerken:
Alle Algorithmen sind als *compare-exchange* Algorithmen verwirklicht, d.h. die Ordnung der Elemente wird allein durch Schlüsselvergleiche bestimmt und allein durch Schlüsselvertauschungen verändert.
Unter dieser Zusatzvoraussetzung werden die auf zyklische Shift–Operationen basierenden Algorithmen BOTTOM-UP-HEAPSORT als auch MDR-HEAPSORT allerdings ungerecht bewertet. Um k Elemente zyklisch zu bewegen, gäbe es so $3(k-1)$ Zuweisungen für die $k-1$ Tauschoperationen statt der tatsächlich nur notwendigen $k+1$ Zuweisungen. Zum Ausgleich dieser Unfairnis kann man die erhaltene Vertauschungszahl letztendlich noch durch 3 dividieren und einen Wert von $2/3(n+\lfloor n/2\rfloor-2)$ hinzufügen ($n+\lfloor n/2\rfloor-2$ *ReHeap*s (vergl. Satz 24) mit jeweils einer Abweichung von ca. 2/3 durch die Addition bzw Subtraktion von 1). Soll sich die Neubewertung der Vertauschungsanzahl für BOTTOM-UP-HEAPSORT und MDR-HEAPSORT auf die Synchronisation der Algorithmen auswirken, so ist es notwendig, die Prozeduren *Interchange* um einen entsprechenden Ausgleich zu erweitern.
Zu den beiden HEAPSORT–Varianten ist außerdem anzumerken, daß der Pfad, der sich durch *LeafSearch* ergibt, mitverwaltet wird.
SHELLSORT benutzt die den worst-case von $O(n^{3/2})$ garantierende rückwärtig zu betrachtende Abstandsfolge $h_k = 3h_{k-1}+1$ mit dem Startwert $h_0 = 1$ unter dem Hinweis auf Sedgewick (1985).
CLEVER-QUICKSORT ist entsprechend der Beschreibung von Sedgewick(1977) implementiert. An das Ende der Rekursion, d.h. wenn weniger als drei Elemente zu vergleichen sind, steht ein effizienter Sortieralgorithmus (3–SORT) für drei Elemente (vergl. Wegener (1995)). Die durch die Auswahl des gesuchten Pivotelemente benötigten Vergleiche werden während der Partitionierung nicht mehr mit dem Pivotelement verglichen.

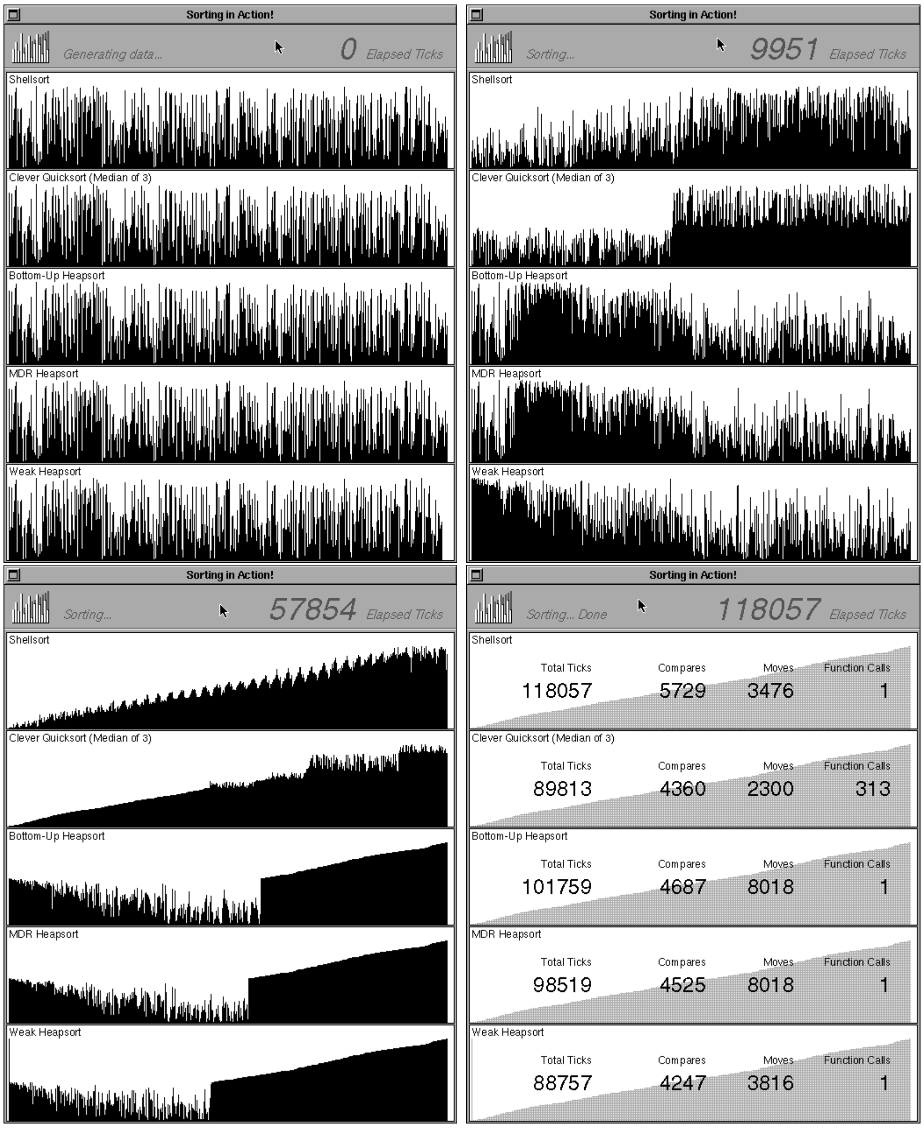

Abbildung 32: Die Sortierverfahren im Vergleich.

Das im Quelltext angegebene Nutzungs– und Kopierrecht sei an dieser Stelle zitiert:

> Author: Julie Zelenski, NeXT Developer Support
> You may freely copy, distribute and reuse the code in this example.
> NeXT disclaims any warranty of any kind, expressed or implied, as
> to its fitness for any particular use.

Demnach wird das Programm für den interessierten Leser bzw. die interessierte Leserin auf dem World–Wide–Web–, kurz WWW– Server des Lehrstuhles Informatik 2 an der Universität Dortmund zur Verfügung gestellt:

> http://ls2-www.informatik.uni-dortmund.de
> (Verweis auf Diplomarbeiten folgen !)

Unter der obigen Adresse ist auch das Postscript–file dieser Diplomarbeit zu finden.
Es läßt sich feststellen, daß WEAK-HEAPSORT mehr Zuweisungen benötigt als die übrigen Algorithmen.
Ist die Bewertung für einen Vergleich wie hier jedoch groß gegenüber der einer Zuweisung, wird der Vorsprung, der sich gegenüber BOTTOM-UP-HEAPSORT und MDR-HEAPSORT aus der äußerst schnellen Aufbauphase ergibt, sozusagen bis über die *Ziellinie* getragen.
Auch SHELLSORT als auch CLEVER-QUICKSORT müssen sich bei einer solch hohen Bewertung der Vergleichsoperation geschlagen geben.
SHELLSORT wird umso stärker je größer der Prozentsatz der Vorsortierung gewählt wird, CLEVER-QUICKSORT entsprechend schlechter.
Die HEAPSORT-Varianten erweisen sich gegenüber diesem Parameter aber eher robust.
Nun zum zweiten Weg, der beschritten worden ist:
Um quantitative Aussagen auch für große Eingabelängen n zu machen, wurden die Algorithmen unabhängig von dem Programm *Sorting in Action* auf einer SUN SPARC–Workstation 4 implementiert und getestet.
Dabei sind die Algorithmen nun nicht mehr *compare-exchange*, d.h. die zyklischen Zuweisungen von BOTTOM-UP–HEAPSORT und MDR-HEAPSORT werden explizit aufgerufen und gezählt. Genauso konnte die Anzahl der Zuweisungen für das in CLEVER-QUICKSORT aufgerufene 3–SORT gesenkt werden. SHELLSORT ist durch den für diese Eingabelängen stärkeren QUICKSORT Algorithmus ersetzt worden. Auch bei QUICKSORT wird das Pivotelement nicht mit sich selber verglichen.
In den Tabellen 6 und 7 werden die Anzahlen von Zuweisungen und Vergleichen der einzelnen Sortieralgorithmen gegenübergestellt. Dabei steigt die Eingabegröße n in Zehntausenderschritten bis 100000 und es werden die über jeweils 20 Versuche gemittelt Werte angegeben.

n	10000	20000	30000	40000	50000
QS	114553	242454	376336	513309	654467
CQS	101574	217377	338523	463960	592687
BUH	174188	368389	569933	776748	987426
MDR	174188	368389	569933	776748	987426
WHS	205735	441324	688833	943430	1202796
n	60000	70000	80000	90000	100000
QS	795508	940302	1084729	1229433	1376276
CQS	721497	853241	983936	1119964	1253806
BUH	1199828	1414884	1633633	1853744	2074946
MDR	1199828	1414884	1633633	1853744	2074946
WHS	1467296	1735408	2005297	2279365	2555401

Tabelle 6: Die Anzahl von Zuweisungen für große Eingabelängen n.

Wiederum benötigt WEAK–HEAPSORT mehr Zuweisungen als die übrigen Algorithmen. Die Anzahl der Zuweisungen liegt um einen fallenden Faktor aus dem Intervall $[1.601, 1.579]$ über der Anzahl der Vergleiche. Er scheint sich bei weiter steigenden n der Grenze 1.5 immer weiter anzunähern. Dies wäre der im Idealfall zu erwartende Wert, da dann jeder zweite in einer $Merge$–Operation durchgeführte Vergleich einen Tausch der Elemente und Teilbäume bewirken würde.
Von Interesse ist es, wie schwer ein Vergleich gegenüber einer Zuweisung gewichtet werden muß, um die beiden QUICKSORT– als auch die beiden HEAPSORT–Varianten zu übertrumpfen.
Seien $WHS_V(n)$ und $WHS_Z(n)$ die Anzahl der Vergleiche bzw. Zuweisungen für WEAK-HEAPSORT und $AS_V(n)$ bzw. $AS_Z(n)$ die eines anderen Sortierverfahrens, dann ergibt der Quotient $(AS_Z(n) - WHS_Z(n))/(WHS_V(n) - AS_V(n))$ das gesuchte Vielfache der Gewichte. Für QUICKSORT liegt dieser Wert für die Tabellen 6 und 7 im Intervall [1.845,2.073], für CLEVER–QUICKSORT im Intervall [6.072,7.007], für BOTTOM–UP–HEAPSORT [4.453,5.867] und für MDR–HEAPSORT im Intervall [6.021,9.152]. Für die ersten beiden Algorithmen ist die Tendenz eher stagnierend, für die zweiten beiden eher steigend.
Die verbrauchte CPU-Zeit der angesprochenen, auf der SPARC 4–Workstation implementierten Algorithmen soll im folgenden als Maßstab der Effizienz be-

n	10000	20000	30000	40000	50000
QS	176769	384709	597237	813263	1027706
CQS	144434	313011	489103	668206	845909
BUH	136666	293334	457233	626707	799787
MDR	133719	287475	448471	614923	785066
WHS	128480	276956	432789	593930	758842
n	60000	70000	80000	90000	100000
QS	1256810	1478399	1721488	1965730	2215907
CQS	1040935	1229685	1436067	1619636	1828910
BUH	974494	1152002	1333304	1515973	1699627
MDR	956929	1131527	1309807	1489559	1670239
WHS	925569	1094977	1267871	1442239	1617747

Tabelle 7: Die Anzahl von Vergleichen für große n.

trachtet werden. Da viele Faktoren wie geschickte Implementierung, effiziente Codegenerierung und evtl. Auslagerung von Hauptspeicherinhalten (sogenanntes swappen) einen starken Einfluß auf die Gesamtlaufzeit haben, sind die Resultate jedoch mit gewisser Vorsicht zu genießen. Zum Beispiel sind alle nicht rekursiven Funktionen für den C-Compiler (gcc) als *inline* deklariert, so daß sich diese Aufrufe nicht negativ bemerkbar machen. Außerdem ist der erzeugte Code nach Möglichkeit optimiert (Option –O2).
Die betrachteten Schlüssel sind auf der einen Seite als zufällig erzeugte natürliche Zahlen (unsigned longs) festgelegt, wobei WEAK–HEAPSORT das Vorzeichenbit als $Reversebit$ verwendet. Auf der anderen Seite stellen die Schlüssel Zeichenketten der Länge 80 dar. Hier sind die Tauschoperationen durch Vertauschung der auf Schlüssel verweisende Zeiger verwirklicht. Die Vergleichsoperation durchsucht die gesamte Zeichenkette unabhängig von dem vielleicht schon ermittelten Vergleichsergebnis.
Dabei steigt die Eingabegröße n wieder in Zehntausenderschritten bis 100000 und es werden die über jeweils 20 Versuche gemittelten Werte angegeben.
Abbildung 33 stellt die ermittelten Ergebnisse graphisch dar.
Gemessen an der CPU–Zeit schlägt WEAK–HEAPSORT sowohl BOTTOM–-UP–HEAPSORT als auch MDR–HEAPSORT mühelos. Dieses Phänomen läßt sich dadurch begründen, daß die $MergeForest$– Prozedur wesentlich schlan-

ker ist als die korrespondierende *ReHeap*–Prozedur. Außerdem ist die Generierungsphase durch die $n-1$ *Merge*–Aufrufe gegenüber den *ReHeap*–Aufrufen bedeutend verkürzt.
Für QUICKSORT und CLEVER–QUICKSORT spricht die schon häufig erwähnte kompakte innere Schleife (Sedgewick (1977)), so werden sie von WEAK-HEAPSORT trotz langer Zeichenketten und großen n's schwerlich geschlagen. Dennoch, für recht komplexe Schlüssel oder vorsortierten Feldern ist WEAK-HEAPSORT diesen Algorithmen vorzuziehen.
Zusammenfassend läßt sich festhalten, daß WEAK–HEAPSORT ein auch für die Praxis sehr zu empfehlendes Sortierverfahren ist.

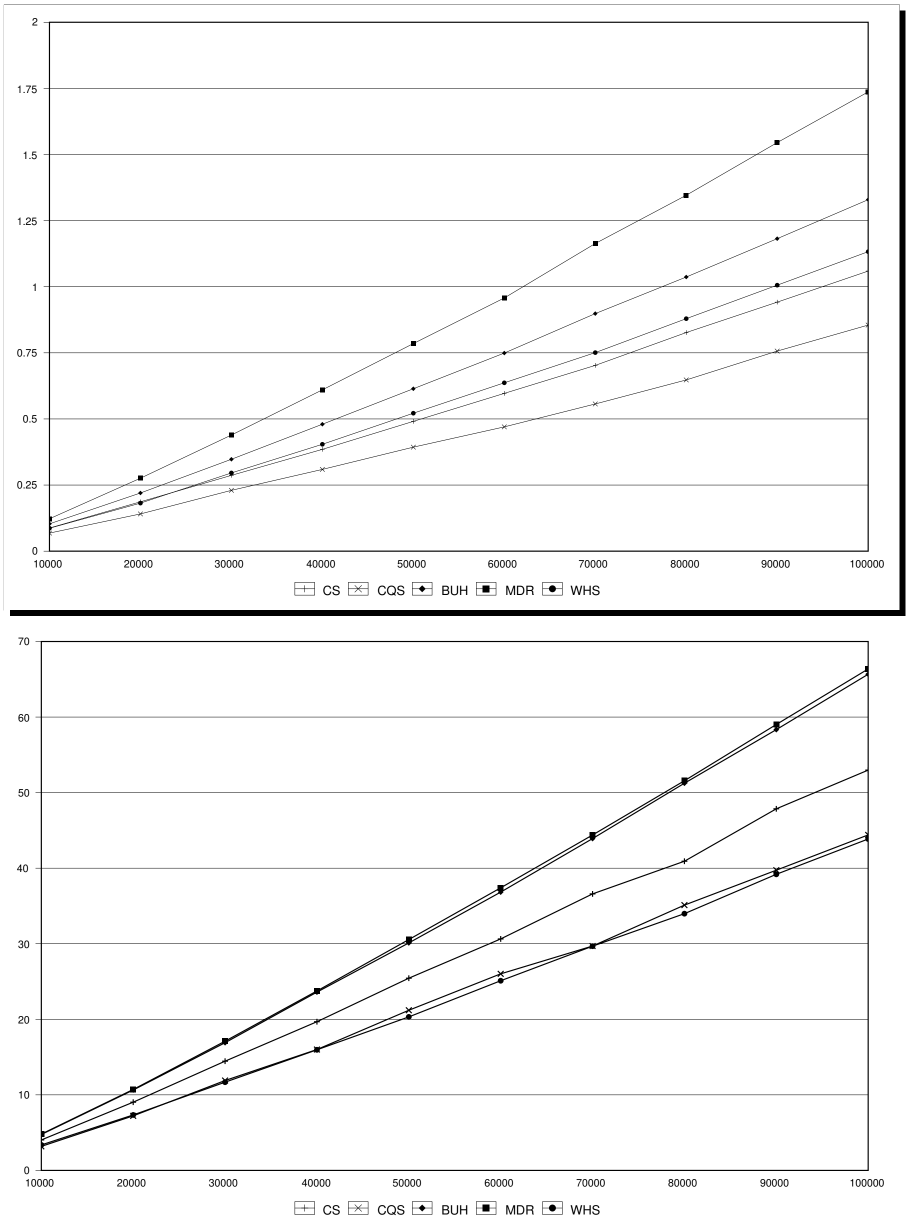

Abbildung 33: CPU-Zeiten für große n: Zahlen und Zeichenketten.

A Eulers Summationsformel

Wer nichts weiß, muß alles glauben.

Marie von Ebner-Eschenbach

Da die Eulersche Summationsformel sowohl die Konstantenberechnung von $\gamma = lim_{n\to\infty}(H_n - \ln n)$ als auch die Approximation von $n!$ ermöglicht, wird dieser allgemeine Ansatz vorgestellt.

Definition 15 *Die Bernoullischen Zahlen B_n, $n \geq 0$, sind rekursiv festgelegt: $B_0 = 1$ und*

$$\sum_{j=0}^{m} \binom{m+1}{j} B_j = 0. \tag{70}$$

Die Bernoullischen Polynome werden beschrieben durch:

$$B_m(x) = \sum_{k=0}^{m} \binom{m}{k} B_k x^{m-k}. \tag{71}$$

Eine Entdeckung von Jakob Bernoulli war die Formel

$$\sum_{k=0}^{n-1} k^m = \frac{1}{m+1} \sum_{k=0}^{m} \binom{m+1}{k} B_k n^{m+1-k}.$$

Sie ergibt sich als Spezialfall der von Leonard Euler entdeckten Formel:

Satz 45 *Sei $f : R \to R$ m-mal stetig differenzierbar. Weiterhin seien $a, b, m \in Z$ mit $a \leq b$ und $m \geq 1$. Dann gilt:*

$$\sum_{k=a}^{b-1} f(k) = \int_a^b f(x)dx + \sum_{k+1}^{m} \frac{B_k}{k!} f^{(k-1)}(x) \mid_a^b + R_m, \tag{72}$$

wobei

$$R_m = (-1)^{m+1} \int_a^b \frac{B_m(x - \lfloor x \rfloor)}{m!} f^{(m)}(x)dx.$$

Falls für $a \leq x \leq b$: $f^{(2m+2)}(x) \geq 0$ und $f^{(2m+4)} \geq 0$ gilt, so ist

$$R_{2m} = \Theta_m \frac{B_{2m+2}}{(2m+2)!} f^{(2m+1)}(x) \mid_a^b$$

für ein $0 < \Theta_m < 1$.
Ist zusätzlich $a = 1$, $b = n$, F die Stammfunktion zu f und $T_k(n) = \frac{B_k}{k!} f^{(k-1)}(n)$, so existiert eine Konstante C mit

$$\sum_{k=1}^{n-1} f(k) = F(n) + C + T_1(n) + \sum_{k=1}^{m} T_{2k}(n) + \Theta_{m,n} T_{2m+2}(n).$$

für $0 < \Theta_{m,n} < 1$.

Beweis: (Graham(1989)). □

Folgerung 16

$$H_{n-1} = \sum_{k=1}^{n-1} \frac{1}{k} = \ln n + \gamma - \frac{1}{2n} - \sum_{k=1}^{m} \frac{B_{2k}}{2kn^{2k}} + \Theta_{m,n} \frac{B_{2m+2}}{(2m+2)n^{2m+2}},$$

wobei $\gamma = 0.57721566\ldots$ *die Eulersche Konstante ist.*

Beweis: (Graham(1989)) Setze $f(x) = 1/x$. Es ist $f^{(m)}(x) = (-1)^m \frac{m!}{x^{m+1}}$ und somit für alle $1 \leq x \leq n$: $f^{(2m)}(x) = \frac{(2m)!}{x^{2m+1}} \geq 0$. Für steigende Werte von m und n kann die obige Formel nach γ aufgelöst werden und γ beliebig genau approximiert werden.□
Für $m = 0$ erhält man $\gamma = lim_{n\to\infty}(H_n - \ln n)$, da sowohl $\frac{B_1}{n}$ als auch $\Theta_{0,n}\frac{B_2}{2n^2}$ gegen 0 konvergieren.
Für $m = 2$ ergibt sich ($H_n = H_{n-1} + 1/n$)

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} + \frac{\Theta_{2,n}}{252n^6}.$$

Folgerung 17 *Sei* $L_n = \sum_{k=1}^{n} \ln k$, *Dann ist*

$$L_{n-1} = n\ln n - n + \rho - \frac{\ln n}{2} + \sum_{k=1}^{m} \frac{B_{2k}}{2k(2k-1)n^{2k-1}} + \Theta_{m,n}\frac{B_{2m+2}}{(2m+2)(2m+1)n^{2m+1}},$$

wobei $e^{\rho} \approx \sqrt{2\pi}$ *die sogenannte Stirlingsche Konstante ist.*

Beweis: (Graham(1989)) Setze $f(x) = \ln x$. Es ist $f^{(m)}(x) = (-1)^{m-1}\frac{(m-1)!}{x^m}$ und somit für alle $1 \leq x \leq n$: $f^{(2m)}(x) = -\frac{(2m-1)!}{x^{2m}} \leq 0$. Dennoch kann der Rest durch den ersten nicht in der Summe auftauchenden Term abgeschätzt werden, da man genauso gut mit $-\ln x$ hätte starten können.
Für feste Werte von m und n kann die obige Formel nach ρ aufgelöst werden und ρ approximiert werden. Es ergibt sich, daß $e^{\rho} \approx \sqrt{2\pi}$ ist.□
Für $m = 2$ ergibt sich ($L_n = L_{n-1} + \ln n$):

$$L_n = \ln n + \rho + \frac{\ln n}{2} + \frac{1}{12n^2} - \frac{1}{360n^3} + \frac{\Theta_{2,n}}{1260n^5}.$$

Da $L_n = \ln\prod_{k=1}^{n} n = \ln n!$ ist, läßt sich über die Anwendung der e-Funktion auf beiden Seiten die bekannte Approximation von $n!$ erzielen.

B Abbildungs– und Tabellenverzeichnis

Zeichnung ist Sprache für die Augen, Sprache Malerei für das Ohr.

Joseph Joubert

Abbildungsverzeichnis

1 Beispiel eines Weak-Heaps. 10
2 Schematische Darstellung der Weak-Heap-Datenstruktur. 10
3 Veranschaulichung der Relation *Gparent*. 11
4 Merge von zwei Weak-Heaps, o.B.d.A. $a[x] \geq a[y]$. 12
5 Modifiziertes Merge: a) $a[x] \geq a[y]$, b) $a[y] > a[x]$. 13
6 Sukzessive Vermengung von m mit den eweiligen Teilbäumen bei $MergeForest$. 15
7 Analyse der Gesamtlänge der Pfade, die durch Gparent definiert sind. 18
8 Die Lage von dem Knoten mit Index m zu dem mit Index x. . . 21
9 Geeignetes Zusammenfassen von Summanden. 22
10 Ein Heap mit $n = 26 = (11010)_2$ Elementen. 60
11 Einbettung eines Weak-Heaps in einen Allgemeinen Heap. 63
12 Boole'sche Variablen beschreiben die Existenz von Blättern. . . . 65
13 Allgemeine Weak-Heap Struktur für $n = 5, 6, 7, 8$. 66
14 Ein nicht zulässiger Weak-Heap mit $n = 6$ Elementen. 67
15 Expansionsausschnitt der Rückwärtsanalyse für die Generierungsphase von HEAPSORT. 70
16 Die Wurzeln vollständiger Teilbüme. 71
17 Korespondenz zwischen $Merge(Gparent(i), i)$ und $MergeUp(j)$. 72
18 Expansionsausschnitt der Rückwärtsanalyse für die Generierungsphase von WEAK–HEAPSORT. 74
19 Beispiel eines rückwärtigen $ReHeap$–Schrittes. 78
20 Expansions der Rückwärtsanalyse für die Auswahlphase von HEAPSORT. 79
21 Ein rückwärtiger $Merge$–Schritt. 81
22 Beispiel eines rückwärtigen $MergeForest^*$–Schrittes. 83
23 Expansions der Rückwärtsanalyse für die Auswahlphase von WEAK–HEAPSORT. 84
24 Unterschiedlichen Modellannahmen: a) Model (M), b) Model (M').105
25 Die Lage der Mengen R_n, L_n und P_n. 110
26 Die Lage von k, m und i in der Menge S. 114
27 Ein Baum, der levelweise eine Summe beschreibt. 118
28 Erster Fall bei der Umformung eines Heaps in einen Weak-Heap. 126

29 Zweiter Fall bei der Umformung eines Heaps in einen Weak-Heap. 127
30 Beispiel der Umformung eines Heaps in einen Weak Heap. 127
31 Das Info–Fenster und das Fenster für die Parameterisierung des Programms *Sorting in Action*. 130
32 Die Sortierverfahren im Vergleich. 132
33 CPU-Zeiten für große n: Zahlen und Zeichenketten. 137

Tabellenverzeichnis

1 Verteilung von Vergleichen nach Dutton. 27
2 Verteilung von Vergleichen beim verbesserten Algorithmus. . . . 28
3 Ausschnitt der Funktionstabelle für $|R_n|$, $|L_n|$ und$|P_n|$. 110
4 Werte für $f(i,j)$, wobei die Zeilen mit i, die Spalten mit j bezeichnet sind. 113
5 Empirische Bestimmung des average-cases von WEAK–HEAP-SORT. 120
6 Die Anzahl von Zuweisungen für große Eingabelängen n. 134
7 Die Anzahl von Vergleichen für große n. 135

C Literaturverzeichnis

Hier sprechen die Stummen und leben die Toten.

Inschrift an einer Bibliothek

Luther hat es verstanden, als er dem Teufel das Tintenfaß an den Kopf geworfen! Nur vor der Tinte fürchtet sich der Teufel; allein damit verjagt man ihn.

Ludwig Börne
Fragmente und Aphorismen 137

Literatur

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1984.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, USA, 1983.

[3] A. Brauer. On a problem of partitions. *Amer. J. Math.*, 64:299–312, 1942.

[4] S. Carlsson. Average-case results on heapsort. *BIT*, 27:2–17, 1987.

[5] S. Carlsson. A variant of heapsort with almost optimal number of comparisons. *Information Processing Letters*, 24(4):247–250, 1987.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[7] R. E. Cypher. A lower bound on the size of shellsort sorting networks. *SIAM Journal on Computing*, 22:1–2, 1993.

[8] E. E. Doberkat. Some observations on the average behavior of heapsort. In *Proc. 21th IEEE Symposium on Foundations of Computer Science (FOCS)*, S. 229–237, 1980.

[9] E. E. Doberkat. Deleting the root of a heap. *Acta Informatica*, 17:245–265, 1982.

[10] E. E. Doberkat. Continous models that are equivalent to randomness for the analysis of many sorting algorithms. *Computing*, 31:11–31, 1983.

[11] E. E. Doberkat. An average case analysis of Floyd's algorithm to construct heaps. *Information and Control*, 61(2):114–131, 1984.

[12] R. D. Dutton. The weak-heap data structure. Technical report, University of Central Florida, Orlando, FL 32816, 1992.

[13] R. D. Dutton. Weak-heap sort. *BIT*, 33:372–381, 1993.

[14] R. Fleischer. A tight lower bound for the worst case of BOTTOM-UP-HEAPSORT. Technical report, Max Planck Inst., Saarbrücken, 1991.

[15] R. W. Floyd. ACM algorithm 245: Treesort 3. *Communications of the ACM*, 7(12):701, 1964.

[16] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, Reading, MA, USA, 1989.

[17] D. H. Greene and D. E. Knuth. *Mathematics for the analysis of algorithms*, volume 1. Progress in computer science, Birkhäuser, 1981.

[18] T. N. Hibbard. A empirical study of minimal storage sorting. *Communications of the ACM*, 6(5):206–213, 1963.

[19] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[20] J. Incerpi and R. Sedgewick. Improved upper bounds on shellsort. *Journal of Computer and System Sciences*, 31:210–224, 1985.

[21] J. Kelly. *Artificial Intelligence A Modern Myth.* Ellis Horwood, Campus 400, Maylands Avenue, Hemel Hempstead, Hertfordshire, HP2 7EZ, 1993.

[22] D. Knuth. *The Art of Computer Programming, Vol. III: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[23] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications.* Text and Monographs in Computer Science. Springer-Verlag, 1993.

[24] C. J. H. McDiarmid and B. A. Reed. Building heaps fast. *Journal of Algorithms*, 10:352–365, 1989.

[25] A. Papernov and G. Stasevich. The worst case in shellsort and related algorithms. *Problems Inform. Transmission*, 1(3):63–75, 1965.

[26] B. Poonen. The worst case in shellsort and related algorithms. *Journal of Algorithms*, 15(1):101–125, 1993.

[27] V. Pratt. *Shellsort and Sorting Networks.* PhD thesis, Stanford University, Garland, New York, 1979.

[28] W. Rudin. *Real and Complex Analysis.* McGraw–Hill, New York (Second Edition), 1974.

[29] R. Schaffer and R. Sedgewick. The analysis of heapsort. *Journal of Algorithms*, 15(1):76–100, 1993.

[30] R. Sedgewick. The analysis of quicksort programs. *Acta Informatica*, 7:327–355, 1977.

[31] R. Sedgewick. *Algorithms*. Addison-Wesley, Reading, 1985.

[32] R. Sedgewick. A new upper bound for shellsort. *Journal of Algorithms*, 2:159–173, 1986.

[33] C. E. Shannon and W. Weaver. *A Mathematical Theory of Communication*. University of Illinois Press, Urbana, Illinois, 1949.

[34] D. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32, 1959.

[35] H. Steinhaus. *One hundred problems in elemantary mathematics (Problems 52,85)*. Pergamon Press, London, 1958.

[36] M. H. van Emden. Increasing the efficiency of QUICKSORT. *Communications of the ACM*, 13:563–567, 1970.

[37] I. Wegener. Datenstrukturen. Lehrstuhl 2, Fachbereich Informatik, Dortmund, 1991. Vorlesungsskript.

[38] I. Wegener. The worst case complexity of McDiarmid and Reed's variant of BOTTOM-UP HEAPSORT is less than $n \log n + 1.1n$. *Information and Computation*, 97(1):86–96, 1992.

[39] I. Wegener. BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT, beating, on an average, QUICKSORT (if n is not very small). *Theoretical Computer Science*, 118:81–98, 1993.

[40] I. Wegener. Komplexitätstheorie. Lehrstuhl 2, Fachbereich Informatik, Dortmund, 1994. Vorlesungsskript.

[41] I. Wegener. Effiziente Algorithmen. Lehrstuhl 2, Fachbereich Informatik, Dortmund, 1995. Vorlesungsskript.

[42] J. W. J. Williams. ACM algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.