SS 2005

Einführung in die Informatik IV

Ernst W. Mayr

Fakultät für Informatik
TU München

http://www14.in.tum.de/lehre/2005SS/info4/index.html.de

24. Juni 2005

1.3 Worst Case-Analyse

Sei A ein Algorithmus. Dann sei

$$T_A(x) := \text{Laufzeit von } A \text{ bei Eingabe } x\,.$$

Diese Funktion ist i.A. zu aufwändig und zu detailliert.
Stattdessen:

$$T_A(n) := \max_{|x|=n} T_A(x) \quad (= \text{maximale Laufzeit bei Eingabelänge } n)$$

1.3 Worst Case-Analyse

Sei A ein Algorithmus. Dann sei

$$T_A(x) := \text{Laufzeit von } A \text{ bei Eingabe } x\,.$$

Diese Funktion ist i.A. zu aufwändig und zu detailliert.
Stattdessen:

$$T_A(n) := \max_{|x|=n} T_A(x) \quad (= \text{maximale Laufzeit bei Eingabelänge } n)$$

1.4 Average Case-Analyse

Oft erscheint die Worst Case-Analyse als zu pessimistisch. Dann:

$$T_A^{\text{ave}}(n) = \frac{\sum_{x;\ |x|=n} T_A(x)}{|\{x;\ |x| = n\}|}$$

oder allgemeiner

$$\begin{aligned} T_A^{\text{ave}}(n) &= \sum T_A(x) \cdot \Pr\{x \mid |x| = n\} \\ &= \mathbf{E}_{|x|=n}\left[T_A(x)\right] , \end{aligned}$$

wobei eine (im Allgemeinen beliebige)
Wahrscheinlichkeitsverteilung zugrunde liegt.

1.4 Average Case-Analyse

Oft erscheint die Worst Case-Analyse als zu pessimistisch. Dann:

$$T_A^{\text{ave}}(n) = \frac{\sum_{x;\,|x|=n} T_A(x)}{|\{x;\ |x| = n\}|}$$

oder allgemeiner

$$\begin{aligned} T_A^{\text{ave}}(n) &= \sum T_A(x) \cdot \text{Pr}\,\{x \mid |x| = n\} \\ &= \mathbf{E}_{|x|=n}\left[T_A(x)\right] \ , \end{aligned}$$

wobei eine (im Allgemeinen beliebige) Wahrscheinlichkeitsverteilung zugrunde liegt.

Bemerkung:
Wir werden Laufzeiten $T_A(n)$ meist nur bis auf einen multiplikativen Faktor genau berechnen, d.h. das genaue Referenzmodell, Fragen der Implementierung, usw. spielen dabei eine eher untergeordnete Rolle.

2. Sortierverfahren

Unter einem Sortierverfahren versteht man ein algorithmisches Verfahren, das als Eingabe eine Folge $a_1, \ldots, a_n$ von n Schlüsseln $\in \Sigma^*$ erhält und als Ausgabe eine auf- oder absteigend sortierte Folge dieser Elemente liefert. Im Folgenden werden wir im Normalfall davon ausgehen, dass die Elemente aufsteigend sortiert werden sollen. Zur Vereinfachung nehmen wir im Normalfall auch an, dass alle Schlüssel paarweise verschieden sind.

Für die betrachteten Sortierverfahren ist natürlich die Anzahl der Schlüsselvergleiche eine untere Schranke für die Laufzeit, und oft ist letztere von der gleichen Größenordnung, d.h.

$$\text{Laufzeit} = O(\text{Anzahl der Schlüsselvergleiche}) .$$

2. Sortierverfahren

Unter einem Sortierverfahren versteht man ein algorithmisches Verfahren, das als Eingabe eine Folge $a_1, \ldots, a_n$ von n Schlüsseln $\in \Sigma^*$ erhält und als Ausgabe eine auf- oder absteigend sortierte Folge dieser Elemente liefert. Im Folgenden werden wir im Normalfall davon ausgehen, dass die Elemente aufsteigend sortiert werden sollen. Zur Vereinfachung nehmen wir im Normalfall auch an, dass alle Schlüssel paarweise verschieden sind.

Für die betrachteten Sortierverfahren ist natürlich die Anzahl der Schlüsselvergleiche eine untere Schranke für die Laufzeit, und oft ist letztere von der gleichen Größenordnung, d.h.

$$\text{Laufzeit} = O(\text{Anzahl der Schlüsselvergleiche}) .$$

2.1 Selection-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus SelectionSort

for $i := n$ **downto** 2 **do**
  $m :=$ Index des maximalen Schlüssels in $A[1..i]$
  vertausche $A[i]$ und $A[m]$
od

Nach Beendigung von SelectionSort ist das Feld A aufsteigend sortiert.

2.1 Selection-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus SelectionSort

for $i := n$ **downto** 2 **do**
  $m :=$ Index des maximalen Schlüssels in $A[1..i]$
  vertausche $A[i]$ und $A[m]$
od

Nach Beendigung von SelectionSort ist das Feld A aufsteigend sortiert.

2.1 Selection-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus SelectionSort

for $i := n$ **downto** 2 **do**
  $m :=$ Index des maximalen Schlüssels in $A[1..i]$
  vertausche $A[i]$ und $A[m]$
od

Nach Beendigung von SelectionSort ist das Feld A aufsteigend sortiert.

2.1 Selection-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus SelectionSort

for $i := n$ **downto** 2 **do**
  $m :=$ Index des maximalen Schlüssels in $A[1..i]$
  vertausche $A[i]$ und $A[m]$
od

Nach Beendigung von SelectionSort ist das Feld A aufsteigend sortiert.

2.1 Selection-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus SelectionSort

for $i := n$ **downto** 2 **do**
  $m :=$ Index des maximalen Schlüssels in $A[1..i]$
  vertausche $A[i]$ und $A[m]$
od

Nach Beendigung von SelectionSort ist das Feld A aufsteigend sortiert.

2.1 Selection-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus SelectionSort

for $i := n$ **downto** 2 **do**
  $m :=$ Index des maximalen Schlüssels in $A[1..i]$
  vertausche $A[i]$ und $A[m]$
od

Nach Beendigung von SelectionSort ist das Feld A aufsteigend sortiert.

2.1 Selection-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus SelectionSort

for $i := n$ **downto** 2 **do**
  $m :=$ Index des maximalen Schlüssels in $A[1..i]$
  vertausche $A[i]$ und $A[m]$
od

Nach Beendigung von SelectionSort ist das Feld A aufsteigend sortiert.

Satz 163

SelectionSort *benötigt zum Sortieren von* n *Elementen genau* $\binom{n}{2}$ *Vergleiche.*

Beweis:

Die Anzahl der Vergleiche (zwischen Schlüsseln bzw. Elementen des Feldes A) zur Bestimmung des maximalen Schlüssels in $A[1..i]$ ist $i-1$.

Damit ergibt sich die Laufzeit von SelectionSort zu

$$T_{\text{SelectionSort}} = \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \binom{n}{2}.$$

□

Satz 163

SelectionSort *benötigt zum Sortieren von* n *Elementen genau* $\binom{n}{2}$ *Vergleiche.*

Beweis:

Die Anzahl der Vergleiche (zwischen Schlüsseln bzw. Elementen des Feldes A) zur Bestimmung des maximalen Schlüssels in $A[1..i]$ ist $i-1$.

Damit ergibt sich die Laufzeit von SelectionSort zu

$$T_{\text{SelectionSort}} = \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \binom{n}{2}.$$

□

Satz 163

SelectionSort *benötigt zum Sortieren von* n *Elementen genau* $\binom{n}{2}$ *Vergleiche.*

Beweis:

Die Anzahl der Vergleiche (zwischen Schlüsseln bzw. Elementen des Feldes A) zur Bestimmung des maximalen Schlüssels in $A[1..i]$ ist $i-1$.

Damit ergibt sich die Laufzeit von SelectionSort zu

$$T_{\text{SelectionSort}} = \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \binom{n}{2}.$$

□

2.2 Insertion-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus InsertionSort

for $i := 2$ **to** n **do**
  $m :=$ Rang $(\in [1..i])$ von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$
  $a := A[i]$
  schiebe $A[m..i-1]$ um eine Position nach rechts
  $A[m] := a$
od

Nach Beendigung von InsertionSort ist das Feld A aufsteigend sortiert.

2.2 Insertion-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus InsertionSort

for $i := 2$ **to** n **do**
  $m :=$ Rang ($\in [1..i]$) von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$
  $a := A[i]$
  schiebe $A[m..i-1]$ um eine Position nach rechts
  $A[m] := a$
od

Nach Beendigung von InsertionSort ist das Feld A aufsteigend sortiert.

2.2 Insertion-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus InsertionSort

for $i := 2$ **to** n **do**
  $m :=$ Rang ($\in [1..i]$) von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$
  $a := A[i]$
  schiebe $A[m..i-1]$ um eine Position nach rechts
  $A[m] := a$
od

Nach Beendigung von InsertionSort ist das Feld A aufsteigend sortiert.

2.2 Insertion-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus InsertionSort

for $i := 2$ **to** n **do**
  $m :=$ Rang ($\in [1..i]$) von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$
  $a := A[i]$
  schiebe $A[m..i-1]$ um eine Position nach rechts
  $A[m] := a$
od

Nach Beendigung von InsertionSort ist das Feld A aufsteigend sortiert.

2.2 Insertion-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus InsertionSort

for $i := 2$ **to** n **do**
  $m :=$ Rang ($\in [1..i]$) von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$
  $a := A[i]$
  schiebe $A[m..i-1]$ um eine Position nach rechts
  $A[m] := a$
od

Nach Beendigung von InsertionSort ist das Feld A aufsteigend sortiert.

2.2 Insertion-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus InsertionSort

for $i := 2$ **to** n **do**
  $m :=$ Rang ($\in [1..i]$) von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$
  $a := A[i]$
  schiebe $A[m..i-1]$ um eine Position nach rechts
  $A[m] := a$
od

Nach Beendigung von InsertionSort ist das Feld A aufsteigend sortiert.

2.2 Insertion-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus InsertionSort

for $i := 2$ **to** n **do**
  $m :=$ Rang ($\in [1..i]$) von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$
  $a := A[i]$
  schiebe $A[m..i-1]$ um eine Position nach rechts
  $A[m] := a$
od

Nach Beendigung von InsertionSort ist das Feld A aufsteigend sortiert.

2.2 Insertion-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus InsertionSort

for $i := 2$ **to** n **do**
  $m :=$ Rang ($\in [1..i]$) von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$
  $a := A[i]$
  schiebe $A[m..i-1]$ um eine Position nach rechts
  $A[m] := a$
od

Nach Beendigung von InsertionSort ist das Feld A aufsteigend sortiert.

2.2 Insertion-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus InsertionSort

for $i := 2$ **to** n **do**
  $m :=$ Rang ($\in [1..i]$) von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$
  $a := A[i]$
  schiebe $A[m..i-1]$ um eine Position nach rechts
  $A[m] := a$
od

Nach Beendigung von InsertionSort ist das Feld A aufsteigend sortiert.

Der Rang von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$ kann trivial mit $i-1$ Vergleichen bestimmt werden. Damit ergibt sich

Satz 164

InsertionSort *benötigt zum Sortieren von n Elementen maximal $\binom{n}{2}$ Vergleiche.*

Beweis:
Übungsaufgabe! □

Der Rang von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$ kann trivial mit $i-1$ Vergleichen bestimmt werden. Damit ergibt sich

Satz 164

InsertionSort *benötigt zum Sortieren von* n *Elementen maximal* $\binom{n}{2}$ *Vergleiche.*

Beweis:

Übungsaufgabe! □

Die Rangbestimmung kann durch binäre Suche verbessert werden. Dabei benötigen wir, um den Rang eines Elementes in einer k-elementigen Menge zu bestimmen, höchstens

$$\lceil \log_2(k+1) \rceil$$

Vergleiche, wie man durch Induktion leicht sieht.

Satz 165

InsertionSort *mit binärer Suche für das Einsortieren benötigt zum Sortieren von n Elementen maximal*

$$n \lceil \mathrm{ld}\, n \rceil$$

Vergleiche.

Beweis:

Die Abschätzung ergibt sich durch einfaches Einsetzen. □

Die Rangbestimmung kann durch binäre Suche verbessert werden. Dabei benötigen wir, um den Rang eines Elementes in einer k-elementigen Menge zu bestimmen, höchstens

$$\lceil \log_2(k+1) \rceil$$

Vergleiche, wie man durch Induktion leicht sieht.

Satz 165

InsertionSort *mit binärer Suche für das Einsortieren benötigt zum Sortieren von n Elementen maximal*

$$n \lceil \mathrm{ld}\, n \rceil$$

Vergleiche.

Beweis:

Die Abschätzung ergibt sich durch einfaches Einsetzen. □

Achtung: Die Laufzeit von InsertionSort ist dennoch auch bei Verwendung von binärer Suche beim Einsortieren im schlechtesten Fall $\Omega(n^2)$, wegen der notwendigen Verschiebung der Feldelemente.

Verwendet man statt des Feldes eine doppelt verkettete Liste, so wird zwar das Einsortieren vereinfacht, es kann jedoch die binäre Suche nicht mehr effizient implementiert werden.

Achtung: Die Laufzeit von InsertionSort ist dennoch auch bei Verwendung von binärer Suche beim Einsortieren im schlechtesten Fall $\Omega(n^2)$, wegen der notwendigen Verschiebung der Feldelemente.

Verwendet man statt des Feldes eine doppelt verkettete Liste, so wird zwar das Einsortieren vereinfacht, es kann jedoch die binäre Suche nicht mehr effizient implementiert werden.

2.3 Merge-Sort

Sei wiederum eine Folge $a_1, \ldots, a_n$ von Schlüsseln im Feld $A[1..n]$ abgespeichert.

Algorithmus MergeSort

proc merge (r, s) **co** sortiere $A[r..s]$ **oc**
  if $s \leq r$ **return fi**
  merge$(r, \lfloor \frac{r+s}{2} \rfloor)$; merge$(\lfloor \frac{r+s}{2} \rfloor + 1, s)$
  verschmelze die beiden sortierten Teilfolgen
end

merge$(1, n)$

2.3 Merge-Sort

Sei wiederum eine Folge $a_1, \ldots, a_n$ von Schlüsseln im Feld $A[1..n]$ abgespeichert.

Algorithmus MergeSort

proc merge (r, s) **co** sortiere $A[r..s]$ **oc**
  if $s \leq r$ **return** **fi**
  merge$(r, \lfloor \frac{r+s}{2} \rfloor)$; merge$(\lfloor \frac{r+s}{2} \rfloor + 1, s)$
  verschmelze die beiden sortierten Teilfolgen
end

merge$(1, n)$

2.3 Merge-Sort

Sei wiederum eine Folge $a_1, \ldots, a_n$ von Schlüsseln im Feld $A[1..n]$ abgespeichert.

Algorithmus MergeSort

proc merge (r, s) **co** sortiere $A[r..s]$ **oc**
  if $s \leq r$ **return fi**
  merge$(r, \lfloor \frac{r+s}{2} \rfloor)$; merge$(\lfloor \frac{r+s}{2} \rfloor + 1, s)$
  verschmelze die beiden sortierten Teilfolgen
end

merge$(1, n)$

2.3 Merge-Sort

Sei wiederum eine Folge $a_1, \ldots, a_n$ von Schlüsseln im Feld $A[1..n]$ abgespeichert.

Algorithmus MergeSort

proc merge (r, s) **co** sortiere $A[r..s]$ **oc**
  if $s \leq r$ **return** **fi**
  merge$(r, \lfloor \frac{r+s}{2} \rfloor)$; merge$(\lfloor \frac{r+s}{2} \rfloor + 1, s)$
  verschmelze die beiden sortierten Teilfolgen
end

merge$(1, n)$

2.3 Merge-Sort

Sei wiederum eine Folge $a_1, \ldots, a_n$ von Schlüsseln im Feld $A[1..n]$ abgespeichert.

Algorithmus MergeSort

proc merge (r, s) **co** sortiere $A[r..s]$ **oc**
  if $s \leq r$ **return** **fi**
  merge$(r, \lfloor \frac{r+s}{2} \rfloor)$; merge$(\lfloor \frac{r+s}{2} \rfloor + 1, s)$
  verschmelze die beiden sortierten Teilfolgen
end

merge$(1, n)$

2.3 Merge-Sort

Sei wiederum eine Folge $a_1, \ldots, a_n$ von Schlüsseln im Feld $A[1..n]$ abgespeichert.

Algorithmus MergeSort

proc merge (r, s) **co** sortiere $A[r..s]$ **oc**
  if $s \leq r$ **return** **fi**
  merge$(r, \lfloor \frac{r+s}{2} \rfloor)$; merge$(\lfloor \frac{r+s}{2} \rfloor + 1, s)$
  verschmelze die beiden sortierten Teilfolgen
end

merge$(1, n)$

2.3 Merge-Sort

Sei wiederum eine Folge $a_1, \ldots, a_n$ von Schlüsseln im Feld $A[1..n]$ abgespeichert.

Algorithmus MergeSort

proc merge (r, s) **co** sortiere $A[r..s]$ **oc**
  if $s \leq r$ **return** **fi**
  merge$(r, \lfloor \frac{r+s}{2} \rfloor)$; merge$(\lfloor \frac{r+s}{2} \rfloor + 1, s)$
  verschmelze die beiden sortierten Teilfolgen
end

merge$(1, n)$

Das Verschmelzen sortierter Teilfolgen $A[r..m]$ und $A[m+1, s]$ funktioniert wie folgt unter Benutzung eines Hilfsfeldes $B[r, s]$:

$i := r$; $j := m + 1$; $k := r$
while $i \leq m$ **and** $j \leq s$ **do**
  if $A[i] < A[j]$ **then** $B[k] := A[i]$; $i := i + 1$
  else $B[k] := A[j]$; $j := j + 1$ **fi**
  $k := k + 1$
od
if $i \leq m$ **then** kopiere $A[i, m]$ nach $B[k, s]$
else kopiere $A[j, s]$ nach $B[k, s]$ **fi**
kopiere $B[r, s]$ nach $A[r, s]$ zurück

Nach Beendigung von MergeSort ist das Feld A aufsteigend sortiert.

Das Verschmelzen sortierter Teilfolgen $A[r..m]$ und $A[m+1, s]$ funktioniert wie folgt unter Benutzung eines Hilfsfeldes $B[r, s]$:

$i := r$; $j := m + 1$; $k := r$
while $i \leq m$ **and** $j \leq s$ **do**
  if $A[i] < A[j]$ **then** $B[k] := A[i]$; $i := i + 1$
  else $B[k] := A[j]$; $j := j + 1$ **fi**
  $k := k + 1$
od
if $i \leq m$ **then** kopiere $A[i, m]$ nach $B[k, s]$
else kopiere $A[j, s]$ nach $B[k, s]$ **fi**
kopiere $B[r, s]$ nach $A[r, s]$ zurück

Nach Beendigung von MergeSort ist das Feld A aufsteigend sortiert.

Das Verschmelzen sortierter Teilfolgen $A[r..m]$ und $A[m+1, s]$ funktioniert wie folgt unter Benutzung eines Hilfsfeldes $B[r, s]$:

$i := r$; $j := m + 1$; $k := r$
while $i \leq m$ **and** $j \leq s$ **do**
  if $A[i] < A[j]$ **then** $B[k] := A[i]$; $i := i + 1$
  else $B[k] := A[j]$; $j := j + 1$ **fi**
  $k := k + 1$
od
if $i \leq m$ **then** kopiere $A[i, m]$ nach $B[k, s]$
else kopiere $A[j, s]$ nach $B[k, s]$ **fi**
kopiere $B[r, s]$ nach $A[r, s]$ zurück

Nach Beendigung von MergeSort ist das Feld A aufsteigend sortiert.

Das Verschmelzen sortierter Teilfolgen $A[r..m]$ und $A[m+1, s]$ funktioniert wie folgt unter Benutzung eines Hilfsfeldes $B[r, s]$:

$i := r$; $j := m + 1$; $k := r$
while $i \leq m$ **and** $j \leq s$ **do**
  if $A[i] < A[j]$ **then** $B[k] := A[i]$; $i := i + 1$
  else $B[k] := A[j]$; $j := j + 1$ **fi**
  $k := k + 1$
od
if $i \leq m$ **then** kopiere $A[i, m]$ nach $B[k, s]$
else kopiere $A[j, s]$ nach $B[k, s]$ **fi**
kopiere $B[r, s]$ nach $A[r, s]$ zurück

Nach Beendigung von MergeSort ist das Feld A aufsteigend sortiert.

Das Verschmelzen sortierter Teilfolgen $A[r..m]$ und $A[m+1, s]$ funktioniert wie folgt unter Benutzung eines Hilfsfeldes $B[r, s]$:

$i := r$; $j := m + 1$; $k := r$
while $i \leq m$ **and** $j \leq s$ **do**
  if $A[i] < A[j]$ **then** $B[k] := A[i]$; $i := i + 1$
  else $B[k] := A[j]$; $j := j + 1$ **fi**
  $k := k + 1$
od
if $i \leq m$ **then** kopiere $A[i, m]$ nach $B[k, s]$
else kopiere $A[j, s]$ nach $B[k, s]$ **fi**
kopiere $B[r, s]$ nach $A[r, s]$ zurück

Nach Beendigung von MergeSort ist das Feld A aufsteigend sortiert.

Das Verschmelzen sortierter Teilfolgen $A[r..m]$ und $A[m+1, s]$ funktioniert wie folgt unter Benutzung eines Hilfsfeldes $B[r, s]$:

$i := r$; $j := m + 1$; $k := r$
while $i \leq m$ **and** $j \leq s$ **do**
  if $A[i] < A[j]$ **then** $B[k] := A[i]$; $i := i + 1$
  else $B[k] := A[j]$; $j := j + 1$ **fi**
  $k := k + 1$
od
if $i \leq m$ **then** kopiere $A[i, m]$ nach $B[k, s]$
else kopiere $A[j, s]$ nach $B[k, s]$ **fi**
kopiere $B[r, s]$ nach $A[r, s]$ zurück

Nach Beendigung von MergeSort ist das Feld A aufsteigend sortiert.

Das Verschmelzen sortierter Teilfolgen $A[r..m]$ und $A[m+1, s]$ funktioniert wie folgt unter Benutzung eines Hilfsfeldes $B[r, s]$:

$i := r$; $j := m + 1$; $k := r$
while $i \leq m$ **and** $j \leq s$ **do**
  if $A[i] < A[j]$ **then** $B[k] := A[i]$; $i := i + 1$
  else $B[k] := A[j]$; $j := j + 1$ **fi**
  $k := k + 1$
od
if $i \leq m$ **then** kopiere $A[i, m]$ nach $B[k, s]$
else kopiere $A[j, s]$ nach $B[k, s]$ **fi**
kopiere $B[r, s]$ nach $A[r, s]$ zurück

Nach Beendigung von MergeSort ist das Feld A aufsteigend sortiert.

Das Verschmelzen sortierter Teilfolgen $A[r..m]$ und $A[m+1, s]$ funktioniert wie folgt unter Benutzung eines Hilfsfeldes $B[r, s]$:

$i := r$; $j := m + 1$; $k := r$
while $i \leq m$ **and** $j \leq s$ **do**
  if $A[i] < A[j]$ **then** $B[k] := A[i]$; $i := i + 1$
  else $B[k] := A[j]$; $j := j + 1$ **fi**
  $k := k + 1$
od
if $i \leq m$ **then** kopiere $A[i, m]$ nach $B[k, s]$
else kopiere $A[j, s]$ nach $B[k, s]$ **fi**
kopiere $B[r, s]$ nach $A[r, s]$ zurück

Nach Beendigung von MergeSort ist das Feld A aufsteigend sortiert.

Das Verschmelzen sortierter Teilfolgen $A[r..m]$ und $A[m+1, s]$ funktioniert wie folgt unter Benutzung eines Hilfsfeldes $B[r, s]$:

$i := r$; $j := m + 1$; $k := r$
while $i \leq m$ **and** $j \leq s$ **do**
  if $A[i] < A[j]$ **then** $B[k] := A[i]$; $i := i + 1$
  else $B[k] := A[j]$; $j := j + 1$ **fi**
  $k := k + 1$
od
if $i \leq m$ **then** kopiere $A[i, m]$ nach $B[k, s]$
else kopiere $A[j, s]$ nach $B[k, s]$ **fi**
kopiere $B[r, s]$ nach $A[r, s]$ zurück

Nach Beendigung von MergeSort ist das Feld A aufsteigend sortiert.

Das Verschmelzen sortierter Teilfolgen $A[r..m]$ und $A[m+1, s]$ funktioniert wie folgt unter Benutzung eines Hilfsfeldes $B[r, s]$:

$i := r$; $j := m + 1$; $k := r$
while $i \leq m$ **and** $j \leq s$ **do**
  if $A[i] < A[j]$ **then** $B[k] := A[i]$; $i := i + 1$
  else $B[k] := A[j]$; $j := j + 1$ **fi**
  $k := k + 1$
od
if $i \leq m$ **then** kopiere $A[i, m]$ nach $B[k, s]$
else kopiere $A[j, s]$ nach $B[k, s]$ **fi**
kopiere $B[r, s]$ nach $A[r, s]$ zurück

Nach Beendigung von MergeSort ist das Feld A aufsteigend sortiert.

Das Verschmelzen sortierter Teilfolgen $A[r..m]$ und $A[m+1, s]$ funktioniert wie folgt unter Benutzung eines Hilfsfeldes $B[r, s]$:

$i := r$; $j := m + 1$; $k := r$
while $i \leq m$ **and** $j \leq s$ **do**
  if $A[i] < A[j]$ **then** $B[k] := A[i]$; $i := i + 1$
  else $B[k] := A[j]$; $j := j + 1$ **fi**
  $k := k + 1$
od
if $i \leq m$ **then** kopiere $A[i, m]$ nach $B[k, s]$
else kopiere $A[j, s]$ nach $B[k, s]$ **fi**
kopiere $B[r, s]$ nach $A[r, s]$ zurück

Nach Beendigung von MergeSort ist das Feld A aufsteigend sortiert.

Satz 166

MergeSort *sortiert ein Feld der Länge* n *mit maximal* $n \cdot \lceil \mathrm{ld}(n) \rceil$ *Vergleichen.*

Beweis:

In jeder Rekursionstiefe werde der Vergleich dem kleineren Element zugeschlagen. Dann erhält jedes Element pro Rekursionstiefe höchstens einen Vergleich zugeschlagen. □

Satz 166

MergeSort *sortiert ein Feld der Länge* n *mit maximal* $n \cdot \lceil \mathrm{ld}(n) \rceil$ *Vergleichen.*

Beweis:

In jeder Rekursionstiefe werde der Vergleich dem kleineren Element zugeschlagen. Dann erhält jedes Element pro Rekursionstiefe höchstens einen Vergleich zugeschlagen. □

2.4 Quick-Sort

Beim Quick-Sort-Verfahren wird in jeder Phase ein Element p der zu sortierenden Folge als Pivot-Element ausgewählt (wie dies geschehen kann, wird noch diskutiert). Dann wird in situ und mit einer linearen Anzahl von Vergleichen die zu sortierende Folge so umgeordnet, dass zuerst alle Elemente $< p$, dann p selbst und schließlich alle Elemente $> p$ kommen. Die beiden Teilfolgen links und rechts von p werden dann mit Quick-Sort rekursiv sortiert (Quick-Sort ist also ein Divide-and-Conquer-Verfahren).

Quick-Sort benötigt im schlechtesten Fall, nämlich wenn als Pivot-Element stets das kleinste oder größte der verbleibenden Elemente ausgewählt wird,

$$\sum_{i=1}^{n-1}(n-i) = \binom{n}{2}$$

Vergleiche.

2.4 Quick-Sort

Beim Quick-Sort-Verfahren wird in jeder Phase ein Element p der zu sortierenden Folge als Pivot-Element ausgewählt (wie dies geschehen kann, wird noch diskutiert). Dann wird in situ und mit einer linearen Anzahl von Vergleichen die zu sortierende Folge so umgeordnet, dass zuerst alle Elemente $< p$, dann p selbst und schließlich alle Elemente $> p$ kommen. Die beiden Teilfolgen links und rechts von p werden dann mit Quick-Sort rekursiv sortiert (Quick-Sort ist also ein Divide-and-Conquer-Verfahren).

Quick-Sort benötigt im schlechtesten Fall, nämlich wenn als Pivot-Element stets das kleinste oder größte der verbleibenden Elemente ausgewählt wird,

$$\sum_{i=1}^{n-1}(n-i) = \binom{n}{2}$$

Vergleiche.

Satz 167

QuickSort *benötigt zum Sortieren eines Feldes der Länge* n *durchschnittlich nur*

$$2\ln(2) \cdot n \operatorname{ld}(n) + O(n)$$

viele Vergleiche.

Beweis:

Siehe Vorlesung Diskrete Strukturen II □

Satz 167

QuickSort *benötigt zum Sortieren eines Feldes der Länge* n *durchschnittlich nur*

$$2\ln(2) \cdot n \operatorname{ld}(n) + O(n)$$

viele Vergleiche.

Beweis:

Siehe Vorlesung Diskrete Strukturen II □

Entscheidend für die Laufzeit von Quick-Sort ist eine „gute“ Wahl des Pivotelements. U.a. werden folgende Varianten verwendet:

1. Nimm stets das letzte Element der (Teil-)Folge als Pivotelement
2. Median-of-3 Verfahren: Wähle den Median (das mittlere Element) des ersten, mittleren und letzten Elements des Arrays
3. Wähle ein zufälliges Element als Pivotelement

Entscheidend für die Laufzeit von Quick-Sort ist eine „gute“ Wahl des Pivotelements. U.a. werden folgende Varianten verwendet:

1. Nimm stets das letzte Element der (Teil-)Folge als Pivotelement
   Nachteil: sehr schlecht bei vorsortierten Arrays!
2. Median-of-3 Verfahren: Wähle den Median (das mittlere Element) des ersten, mittleren und letzten Elements des Arrays
3. Wähle ein zufälliges Element als Pivotelement

Entscheidend für die Laufzeit von Quick-Sort ist eine „gute“ Wahl des Pivotelements. U.a. werden folgende Varianten verwendet:

1. Nimm stets das letzte Element der (Teil-)Folge als Pivotelement
   Nachteil: sehr schlecht bei vorsortierten Arrays!
2. Median-of-3 Verfahren: Wähle den Median (das mittlere Element) des ersten, mittleren und letzten Elements des Arrays
3. Wähle ein zufälliges Element als Pivotelement

Entscheidend für die Laufzeit von Quick-Sort ist eine „gute“ Wahl des Pivotelements. U.a. werden folgende Varianten verwendet:

1. Nimm stets das letzte Element der (Teil-)Folge als Pivotelement
   Nachteil: sehr schlecht bei vorsortierten Arrays!
2. Median-of-3 Verfahren: Wähle den Median (das mittlere Element) des ersten, mittleren und letzten Elements des Arrays
   Analyse Übungsaufgabe!
3. Wähle ein zufälliges Element als Pivotelement

Entscheidend für die Laufzeit von Quick-Sort ist eine „gute“ Wahl des Pivotelements. U.a. werden folgende Varianten verwendet:

1. Nimm stets das letzte Element der (Teil-)Folge als Pivotelement
   Nachteil: sehr schlecht bei vorsortierten Arrays!
2. Median-of-3 Verfahren: Wähle den Median (das mittlere Element) des ersten, mittleren und letzten Elements des Arrays
   Analyse Übungsaufgabe!
3. Wähle ein zufälliges Element als Pivotelement

Entscheidend für die Laufzeit von Quick-Sort ist eine „gute“ Wahl des Pivotelements. U.a. werden folgende Varianten verwendet:

1. Nimm stets das letzte Element der (Teil-)Folge als Pivotelement
   Nachteil: sehr schlecht bei vorsortierten Arrays!
2. Median-of-3 Verfahren: Wähle den Median (das mittlere Element) des ersten, mittleren und letzten Elements des Arrays
   Analyse Übungsaufgabe!
3. Wähle ein zufälliges Element als Pivotelement
   liefert die o.a. durchschnittliche Laufzeit, benötigt aber einen Zufallsgenerator.

2.5 Heap-Sort

Definition 168

Ein Heap ist ein Binärbaum, an dessen Knoten Schlüssel gespeichert sind, so dass gilt:

1. alle inneren Knoten bis auf maximal einen haben genau zwei Kinder;
2. alle Knoten mit weniger als zwei Kindern (also insbesondere die Blätter) befinden sich auf der untersten oder der zweituntersten Schicht;
3. die unterste Schicht ist von links nach rechts aufgefüllt;
4. für jeden Knoten (mit Ausnahme der Wurzel) gilt, dass sein Schlüssel kleiner ist als der seines Vaters (Heap-Bedingung).

2.5 Heap-Sort

Definition 168

Ein Heap ist ein Binärbaum, an dessen Knoten Schlüssel gespeichert sind, so dass gilt:

1. alle inneren Knoten bis auf maximal einen haben genau zwei Kinder;
2. alle Knoten mit weniger als zwei Kindern (also insbesondere die Blätter) befinden sich auf der untersten oder der zweituntersten Schicht;
3. die unterste Schicht ist von links nach rechts aufgefüllt;
4. für jeden Knoten (mit Ausnahme der Wurzel) gilt, dass sein Schlüssel kleiner ist als der seines Vaters (Heap-Bedingung).

2.5 Heap-Sort

Definition 168

Ein Heap ist ein Binärbaum, an dessen Knoten Schlüssel gespeichert sind, so dass gilt:

1. alle inneren Knoten bis auf maximal einen haben genau zwei Kinder;
2. alle Knoten mit weniger als zwei Kindern (also insbesondere die Blätter) befinden sich auf der untersten oder der zweituntersten Schicht;
3. die unterste Schicht ist von links nach rechts aufgefüllt;
4. für jeden Knoten (mit Ausnahme der Wurzel) gilt, dass sein Schlüssel kleiner ist als der seines Vaters (Heap-Bedingung).

2.5 Heap-Sort

Definition 168

Ein Heap ist ein Binärbaum, an dessen Knoten Schlüssel gespeichert sind, so dass gilt:

1. alle inneren Knoten bis auf maximal einen haben genau zwei Kinder;
2. alle Knoten mit weniger als zwei Kindern (also insbesondere die Blätter) befinden sich auf der untersten oder der zweituntersten Schicht;
3. die unterste Schicht ist von links nach rechts aufgefüllt;
4. für jeden Knoten (mit Ausnahme der Wurzel) gilt, dass sein Schlüssel kleiner ist als der seines Vaters (Heap-Bedingung).

2.5 Heap-Sort

Definition 168

Ein Heap ist ein Binärbaum, an dessen Knoten Schlüssel gespeichert sind, so dass gilt:

1. alle inneren Knoten bis auf maximal einen haben genau zwei Kinder;
2. alle Knoten mit weniger als zwei Kindern (also insbesondere die Blätter) befinden sich auf der untersten oder der zweituntersten Schicht;
3. die unterste Schicht ist von links nach rechts aufgefüllt;
4. für jeden Knoten (mit Ausnahme der Wurzel) gilt, dass sein Schlüssel kleiner ist als der seines Vaters (Heap-Bedingung).

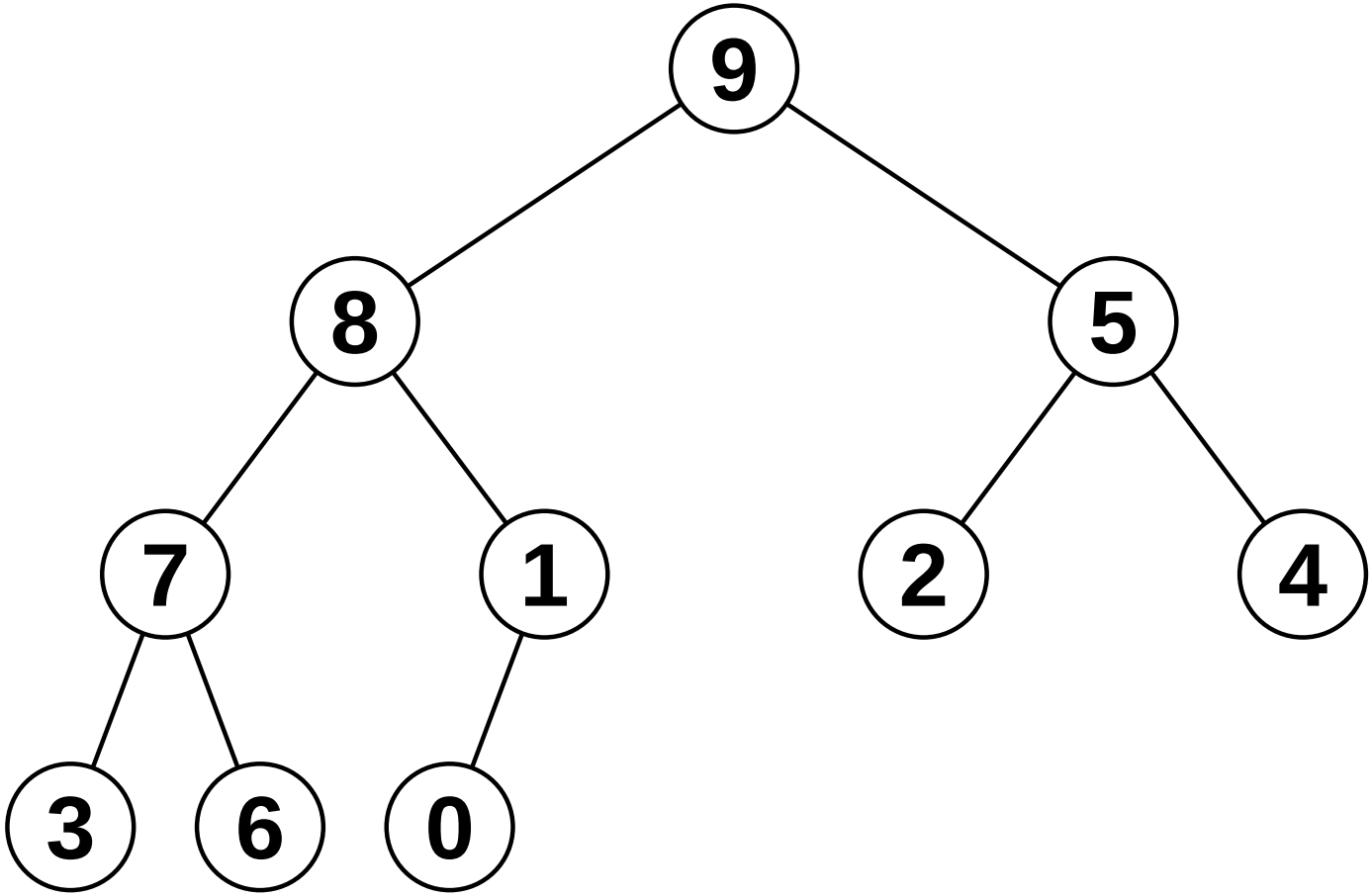
Bemerkungen:

1. Die hier definierte Variante ist ein max-Heap.
2. Die Bezeichnung heap wird in der Algorithmentheorie auch allgemeiner für Prioritätswarteschlangen benutzt!

Bemerkungen:

1. Die hier definierte Variante ist ein max-Heap.
2. Die Bezeichnung heap wird in der Algorithmentheorie auch allgemeiner für Prioritätswarteschlangen benutzt!

Beispiel 169



Der Algorithmus HeapSort besteht aus zwei Phasen.

1. In der ersten Phase wird aus der unsortierten Folge von n Elementen ein Heap gemäß Definition aufgebaut.
2. In der zweiten Phase wird dieser Heap ausgegeben, d.h. ihm wird n-mal jeweils das größte Element entnommen (das ja an der Wurzel steht), dieses Element wird in die zu sortierende Folge aufgenommen und die Heap-Eigenschaften wird wieder hergestellt.

Der Algorithmus HeapSort besteht aus zwei Phasen.

1. In der ersten Phase wird aus der unsortierten Folge von n Elementen ein Heap gemäß Definition aufgebaut.
2. In der zweiten Phase wird dieser Heap ausgegeben, d.h. ihm wird n-mal jeweils das größte Element entnommen (das ja an der Wurzel steht), dieses Element wird in die zu sortierende Folge aufgenommen und die Heap-Eigenschaften wird wieder hergestellt.

Betrachten wir nun zunächst den Algorithmus REHEAP zur Korrektur der Datenstruktur, falls die Heap-Bedingung höchstens an der Wurzel verletzt ist.

Algorithmus REHEAP

sei v die Wurzel des Heaps;
while Heap-Eigenschaft in v nicht erfüllt **do**
  sei v' das Kind von v mit dem größeren Schlüssel
  vertausche die Schlüssel in v und v'
  $v := v'$
od

Zweite Phase von HeapSort

for $i := n$ **downto** 1 do
  sei r die Wurzel des Heaps
  sei k der in r gespeicherte Schlüssel
  $A[i] := k$
  sei b das rechteste Blatt in der untersten Schicht des Heaps
  kopiere den Schlüssel von b in die Wurzel r
  entferne das Blatt b
  Reheap
od

Nach Beendigung der zweiten Phase von HeapSort ist das Feld $A[1..n]$ aufsteigend sortiert.

Zweite Phase von HeapSort

for $i := n$ **downto** 1 do
  sei r die Wurzel des Heaps
  sei k der in r gespeicherte Schlüssel
  $A[i] := k$
  sei b das rechteste Blatt in der untersten Schicht des Heaps
  kopiere den Schlüssel von b in die Wurzel r
  entferne das Blatt b
  Reheap
od

Nach Beendigung der zweiten Phase von HeapSort ist das Feld $A[1..n]$ aufsteigend sortiert.

Zweite Phase von HeapSort

for $i := n$ **downto** 1 do
  sei r die Wurzel des Heaps
  sei k der in r gespeicherte Schlüssel
  $A[i] := k$
  sei b das rechteste Blatt in der untersten Schicht des Heaps
  kopiere den Schlüssel von b in die Wurzel r
  entferne das Blatt b
  Reheap
od

Nach Beendigung der zweiten Phase von HeapSort ist das Feld $A[1..n]$ aufsteigend sortiert.

Zweite Phase von HeapSort

for $i := n$ **downto** 1 do
  sei r die Wurzel des Heaps
  sei k der in r gespeicherte Schlüssel
  $A[i] := k$
  sei b das rechteste Blatt in der untersten Schicht des Heaps
  kopiere den Schlüssel von b in die Wurzel r
  entferne das Blatt b
  Reheap
od

Nach Beendigung der zweiten Phase von HeapSort ist das Feld $A[1..n]$ aufsteigend sortiert.

Zweite Phase von HeapSort

for $i := n$ **downto** 1 do
  sei r die Wurzel des Heaps
  sei k der in r gespeicherte Schlüssel
  $A[i] := k$
  sei b das rechteste Blatt in der untersten Schicht des Heaps
  kopiere den Schlüssel von b in die Wurzel r
  entferne das Blatt b
  Reheap
od

Nach Beendigung der zweiten Phase von HeapSort ist das Feld $A[1..n]$ aufsteigend sortiert.

Zweite Phase von HeapSort

for $i := n$ **downto** 1 do
  sei r die Wurzel des Heaps
  sei k der in r gespeicherte Schlüssel
  $A[i] := k$
  sei b das rechteste Blatt in der untersten Schicht des Heaps
  kopiere den Schlüssel von b in die Wurzel r
  entferne das Blatt b
  Reheap
od

Nach Beendigung der zweiten Phase von HeapSort ist das Feld $A[1..n]$ aufsteigend sortiert.

Zweite Phase von HeapSort

for $i := n$ **downto** 1 do
  sei r die Wurzel des Heaps
  sei k der in r gespeicherte Schlüssel
  $A[i] := k$
  sei b das rechteste Blatt in der untersten Schicht des Heaps
  kopiere den Schlüssel von b in die Wurzel r
  entferne das Blatt b
  Reheap
od

Nach Beendigung der zweiten Phase von HeapSort ist das Feld $A[1..n]$ aufsteigend sortiert.

Zweite Phase von HeapSort

for $i := n$ **downto** 1 do
  sei r die Wurzel des Heaps
  sei k der in r gespeicherte Schlüssel
  $A[i] := k$
  sei b das rechteste Blatt in der untersten Schicht des Heaps
  kopiere den Schlüssel von b in die Wurzel r
  entferne das Blatt b
  Reheap
od

Nach Beendigung der zweiten Phase von HeapSort ist das Feld $A[1..n]$ aufsteigend sortiert.

Zweite Phase von HeapSort

for $i := n$ **downto** 1 do
  sei r die Wurzel des Heaps
  sei k der in r gespeicherte Schlüssel
  $A[i] := k$
  sei b das rechteste Blatt in der untersten Schicht des Heaps
  kopiere den Schlüssel von b in die Wurzel r
  entferne das Blatt b
  Reheap
od

Nach Beendigung der zweiten Phase von HeapSort ist das Feld $A[1..n]$ aufsteigend sortiert.

Zweite Phase von HeapSort

for $i := n$ **downto** 1 do
  sei r die Wurzel des Heaps
  sei k der in r gespeicherte Schlüssel
  $A[i] := k$
  sei b das rechteste Blatt in der untersten Schicht des Heaps
  kopiere den Schlüssel von b in die Wurzel r
  entferne das Blatt b
  Reheap
od

Nach Beendigung der zweiten Phase von HeapSort ist das Feld $A[1..n]$ aufsteigend sortiert.

Zweite Phase von HeapSort

for $i := n$ **downto** 1 do
  sei r die Wurzel des Heaps
  sei k der in r gespeicherte Schlüssel
  $A[i] := k$
  sei b das rechteste Blatt in der untersten Schicht des Heaps
  kopiere den Schlüssel von b in die Wurzel r
  entferne das Blatt b
  Reheap
od

Nach Beendigung der zweiten Phase von HeapSort ist das Feld $A[1..n]$ aufsteigend sortiert.

In der ersten Phase von HeapSort müssen wir mit den gegebenen n Schlüsseln einen Heap erzeugen.

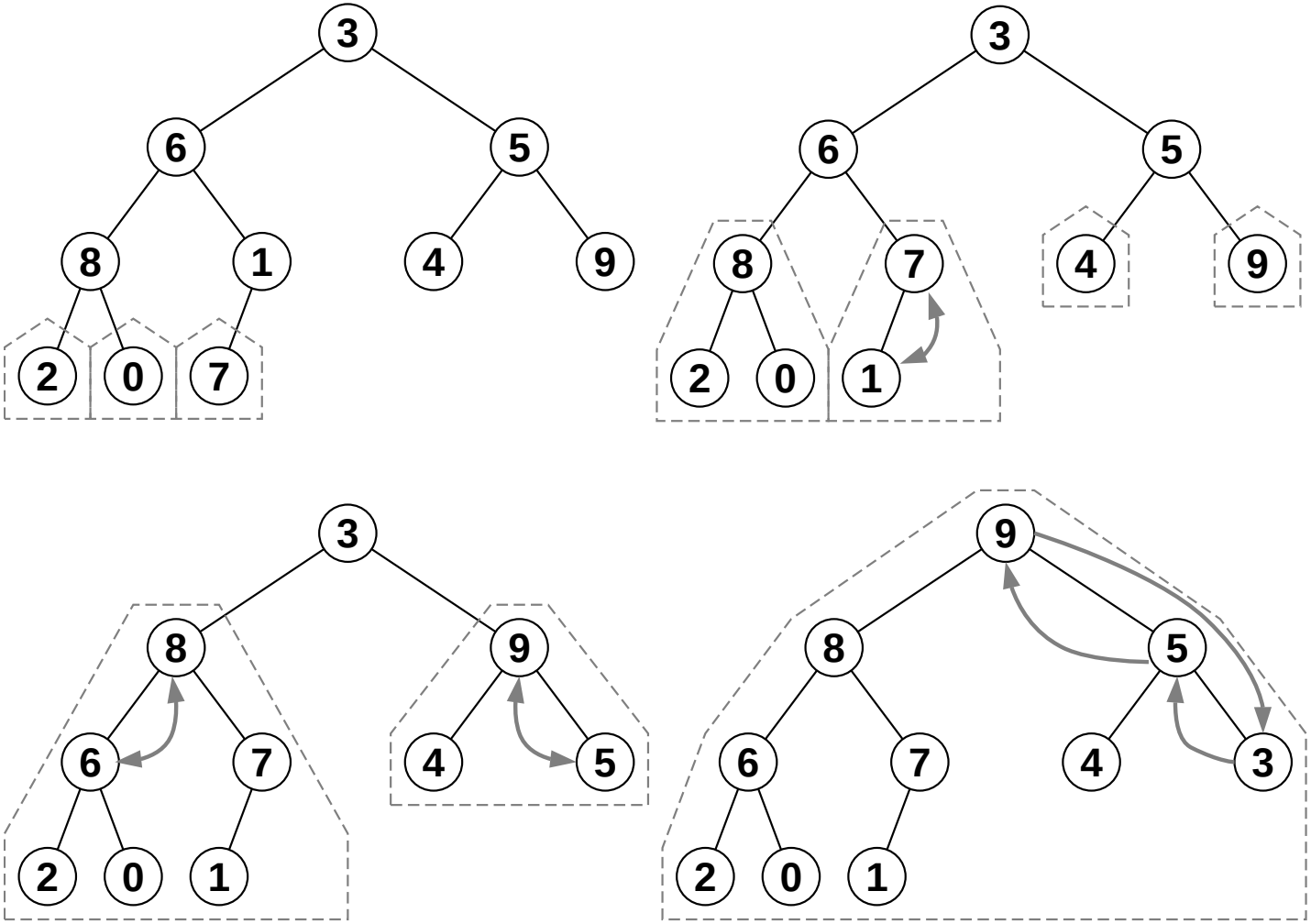
Wir tun dies iterativ, indem wir aus zwei bereits erzeugten Heaps und einem weiteren Schlüssel einen neuen Heap formen, indem wir einen neuen Knoten erzeugen, der die Wurzel des neuen Heaps wird. Diesem neuen Knoten ordnen wir zunächst den zusätzlichen Schlüssel zu und und machen die beiden alten Heaps zu seinen Unterbäumen. Damit ist die Heap-Bedingung höchstens an der Wurzel des neuen Baums verletzt, was wir durch Ausführung der Reheap-Operation korrigieren können.

In der ersten Phase von HeapSort müssen wir mit den gegebenen n Schlüsseln einen Heap erzeugen.

Wir tun dies iterativ, indem wir aus zwei bereits erzeugten Heaps und einem weiteren Schlüssel einen neuen Heap formen, indem wir einen neuen Knoten erzeugen, der die Wurzel des neuen Heaps wird. Diesem neuen Knoten ordnen wir zunächst den zusätzlichen Schlüssel zu und und machen die beiden alten Heaps zu seinen Unterbäumen. Damit ist die Heap-Bedingung höchstens an der Wurzel des neuen Baums verletzt, was wir durch Ausführung der Reheap-Operation korrigieren können.

Beispiel 170

[Initialisierung des Heaps]



Lemma 171

Die Reheap-Operation erfordert höchstens $O($Tiefe des Heaps$)$ Schritte.

Beweis:

Reheap führt pro Schicht des Heaps nur konstant viele Schritte aus. □

Lemma 171

Die Reheap-Operation erfordert höchstens $O($Tiefe des Heaps$)$ Schritte.

Beweis:

Reheap führt pro Schicht des Heaps nur konstant viele Schritte aus. □

Lemma 172

Die Initialisierung des Heaps in der ersten Phase von HeapSort *benötigt nur $O(n)$ Schritte.*

Beweis:

Sei d die Tiefe (Anzahl der Schichten) des (n-elementigen) Heaps. Die Anzahl der Knoten in Tiefe i ist $\leq 2^i$. Wenn ein solcher Knoten beim inkrementellen Aufbau des Heaps als Wurzel hinzugefügt wird, erfordert die Reheap-Operation $\leq d - i$ Schritte, insgesamt werden also

$$\leq \sum_{i=0}^{d} (d-i)2^i = O(n)$$

Schritte benötigt. □

Lemma 172

Die Initialisierung des Heaps in der ersten Phase von HeapSort *benötigt nur $O(n)$ Schritte.*

Beweis:

Sei d die Tiefe (Anzahl der Schichten) des (n-elementigen) Heaps. Die Anzahl der Knoten in Tiefe i ist $\leq 2^i$. Wenn ein solcher Knoten beim inkrementellen Aufbau des Heaps als Wurzel hinzugefügt wird, erfordert die Reheap-Operation $\leq d - i$ Schritte, insgesamt werden also

$$\leq \sum_{i=0}^{d} (d-i)2^i = O(n)$$

Schritte benötigt. □

Satz 173

HeapSort *benötigt maximal* $O(n \log n)$ *Schritte.*

Beweis:

In der zweiten Phase werden $< n$ Reheap-Operationen auf Heaps der Tiefe $\leq \log n$ durchgeführt. □

Bemerkung:

1. Eine genauere Analyse ergibt eine Schranke von $2n \operatorname{ld}(n) + o(n)$.
2. Carlsson hat eine Variante von HeapSort beschrieben, die mit $n \operatorname{ld} n + O(n \log \log n)$ Vergleichen auskommt.

Satz 173

HeapSort *benötigt maximal* $O(n \log n)$ *Schritte.*

Beweis:

In der zweiten Phase werden $< n$ Reheap-Operationen auf Heaps der Tiefe $\leq \log n$ durchgeführt. □

Bemerkung:

1. Eine genauere Analyse ergibt eine Schranke von $2n \operatorname{ld}(n) + o(n)$.
2. Carlsson hat eine Variante von HeapSort beschrieben, die mit $n \operatorname{ld} n + O(n \log \log n)$ Vergleichen auskommt.

Satz 173

HeapSort *benötigt maximal* $O(n \log n)$ *Schritte.*

Beweis:

In der zweiten Phase werden $< n$ Reheap-Operationen auf Heaps der Tiefe $\leq \log n$ durchgeführt. □

Bemerkung:

1. Eine genauere Analyse ergibt eine Schranke von $2n \operatorname{ld}(n) + o(n)$.
2. Carlsson hat eine Variante von HeapSort beschrieben, die mit $n \operatorname{ld} n + O(n \log \log n)$ Vergleichen auskommt.

Satz 173

HeapSort *benötigt maximal* $O(n \log n)$ *Schritte.*

Beweis:

In der zweiten Phase werden $< n$ Reheap-Operationen auf Heaps der Tiefe $\leq \log n$ durchgeführt. □

Bemerkung:

1. Eine genauere Analyse ergibt eine Schranke von $2n \operatorname{ld}(n) + o(n)$.
2. Carlsson hat eine Variante von HeapSort beschrieben, die mit $n \operatorname{ld} n + O(n \log \log n)$ Vergleichen auskommt.

Satz 173

HeapSort *benötigt maximal* $O(n \log n)$ *Schritte.*

Beweis:

In der zweiten Phase werden $< n$ Reheap-Operationen auf Heaps der Tiefe $\leq \log n$ durchgeführt. □

Bemerkung:

1. Eine genauere Analyse ergibt eine Schranke von $2n \operatorname{ld}(n) + o(n)$.
2. Carlsson hat eine Variante von HeapSort beschrieben, die mit $n \operatorname{ld} n + O(n \log \log n)$ Vergleichen auskommt.