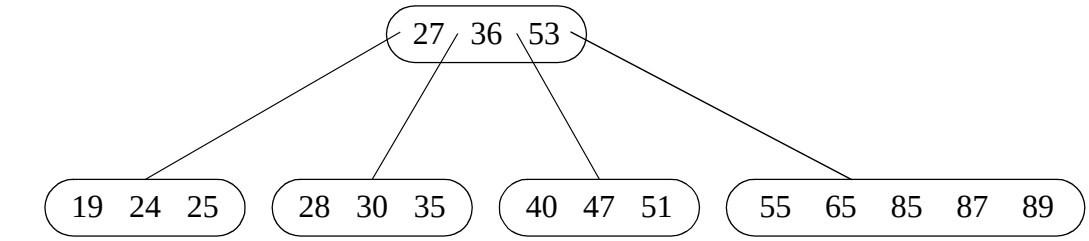
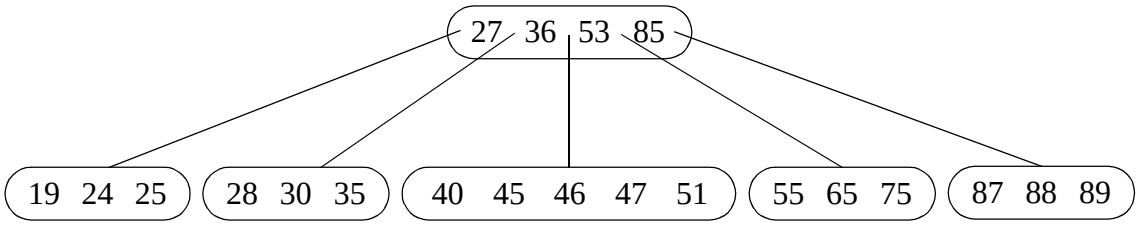
Musterlösungen
zur Nachklausur
„1663 Datenstrukturen“

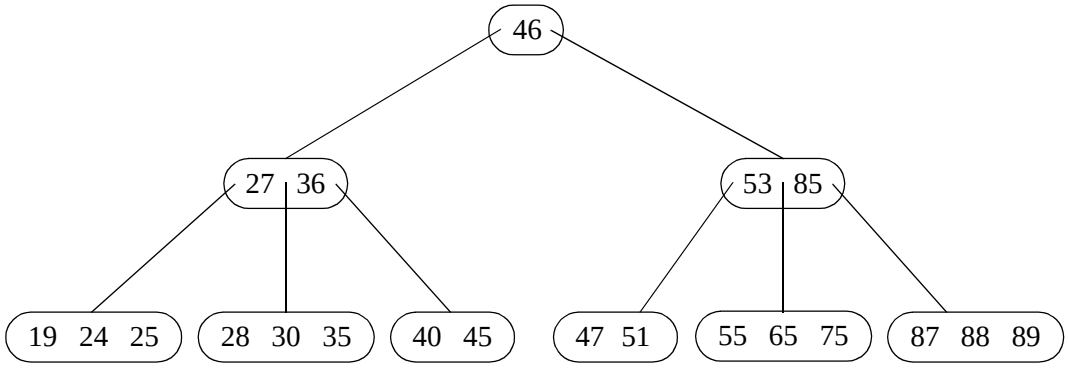
23. September 2000

Aufgabe 1

(a)

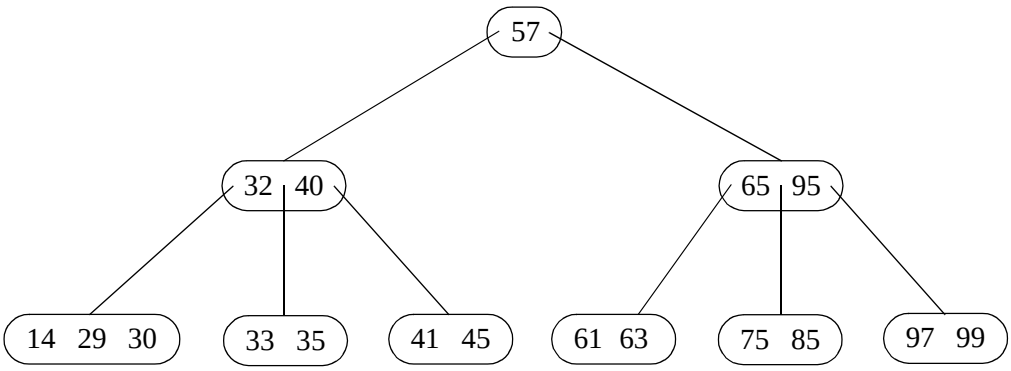

27 36 53
19 24 25
28 30 35
40 47 51
55 65 85 87 89


27 36 53 85
19 24 25
28 30 35
40 45 46 47 51
55 65 75
87 88 89

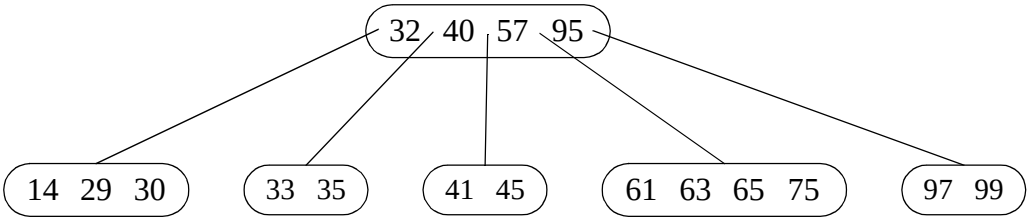

46
27 36
53 85
19 24 25
28 30 35
40 45
47 51
55 65 75
87 88 89

(b)

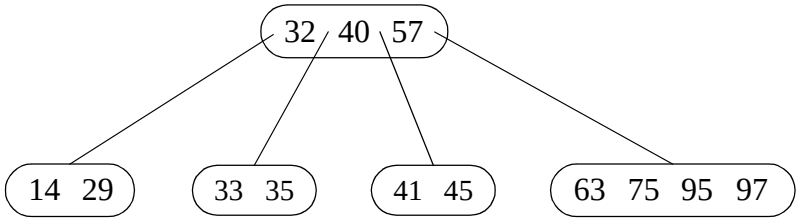
Das Löschen von 98 ist problemlos. Löschen von 50 führt zu einem Underflow mit einer *balance*-Operation:



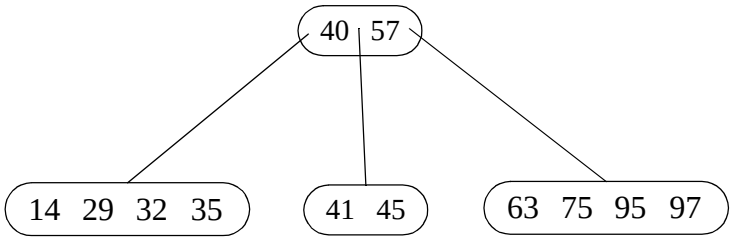
Löschen von 85 führt zu einem Underflow und erfordert zwei *merge*-Operationen:



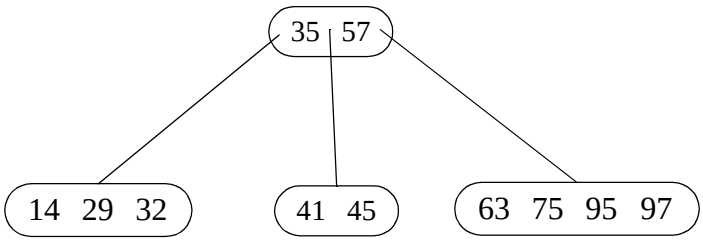
Löschen von 65, 30 und 61 ist problemlos. Dann führt das Löschen von 99 zu einem Underflow und erfordert eine *merge*-Operation:



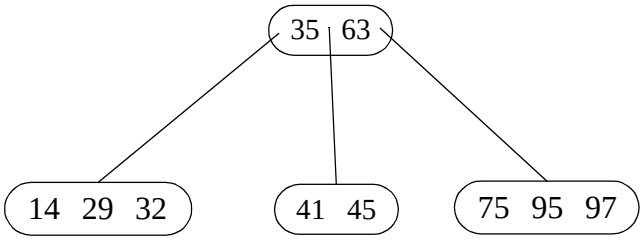
Löschen von 33 führt zu einem Underflow mit *merge*-Operation:



Löschen von 40: Ersetzen durch 41 mit anschließender *balance*-Operation:



Löschen von 57: Ersetzen durch 63:



Aufgabe 2

(a)

Unter dem Begriff „Heap-Eigenschaft“ versteht man die Eigenschaft, dass der Schlüsselwert eines jeden Knotens größer oder gleich den Schlüsselwerten seiner Söhne ist, falls diese existieren. D.h., für jeden Knoten i in der Array-Repräsentation A eines Heaps gilt, dass $A[i] \geq A[2i]$ und $A[i] \geq A[2i + 1]$.

(b)

Sei n die Anzahl der Elemente (Knoten) eines Heaps. Aus dem Kurs wissen wir, dass ein binärer Baum der Höhe h maximal 2^{h+1}-1 Knoten besitzt. Er ist dann vollständig ausgeglichen. Andererseits muß die Ebene mit genau der Höhe h mindestens einen Knoten besitzen. Daher ergibt sich für einen Heap der Höhe h mit n Knoten: $2^h \leq n \leq 2^{h+1}$-1.

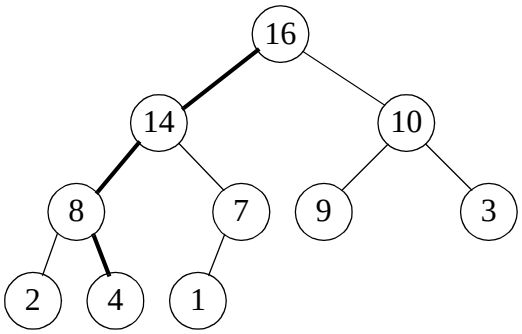
(c)

Das kleinste Element der gesamten zu sortierenden Folge muß sich in einem Blatt des Heaps befinden. Wenn sich dieses Element im Innern des Heaps befinden würde, hätte es Söhne, deren Schlüsselwerte gemäß der Heap-Eigenschaft kleiner sind. Dann könnte dieses Element aber nicht das kleinste sein.

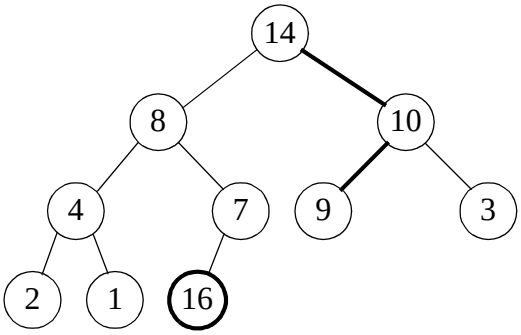
(d)

Weil die Elemente $A[\lfloor n/2 \rfloor + 1], ..., A[n]$ Blätter des Heaps sind, bildet jedes Element bereits einen einelementigen Heap. Die Reihenfolge (nämlich von hinten nach vorne / von unten nach oben), in der die weiteren Elemente verarbeitet werden, stellt sicher, daß die Teilbäume, deren Wurzeln die Söhne eines Knotens i sind, bereits Heaps sind, bevor *reheap* bezüglich i angewendet wird.

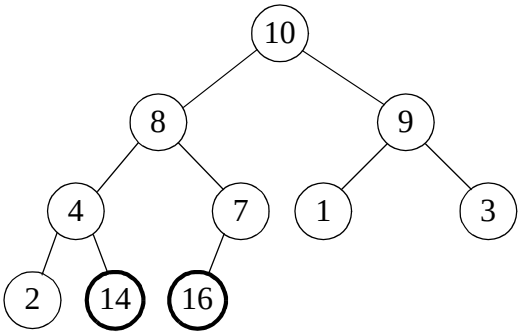
(e)



Ausgangsheap



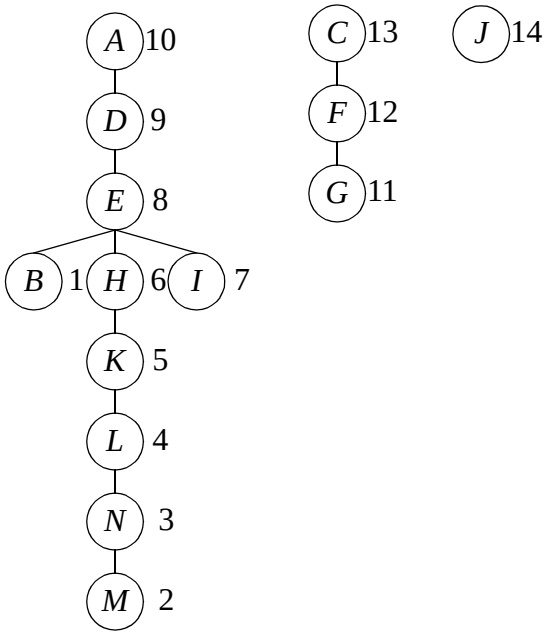
1. Schritt



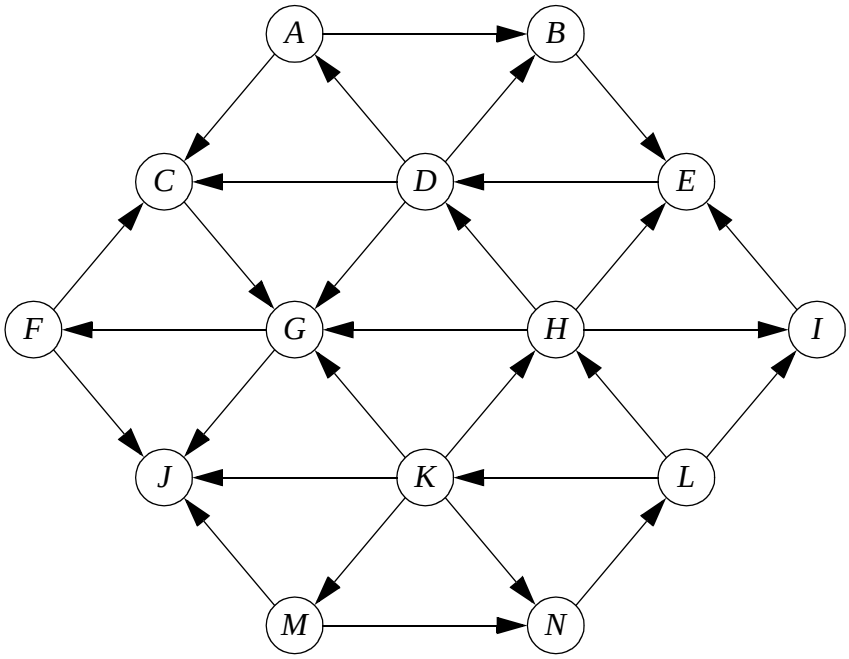
2. Schritt

Aufgabe 3

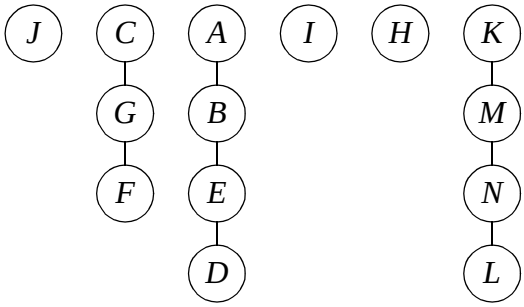
Zunächst werden die Depth-First-Spannbäume samt Knotennummerierung berechnet:



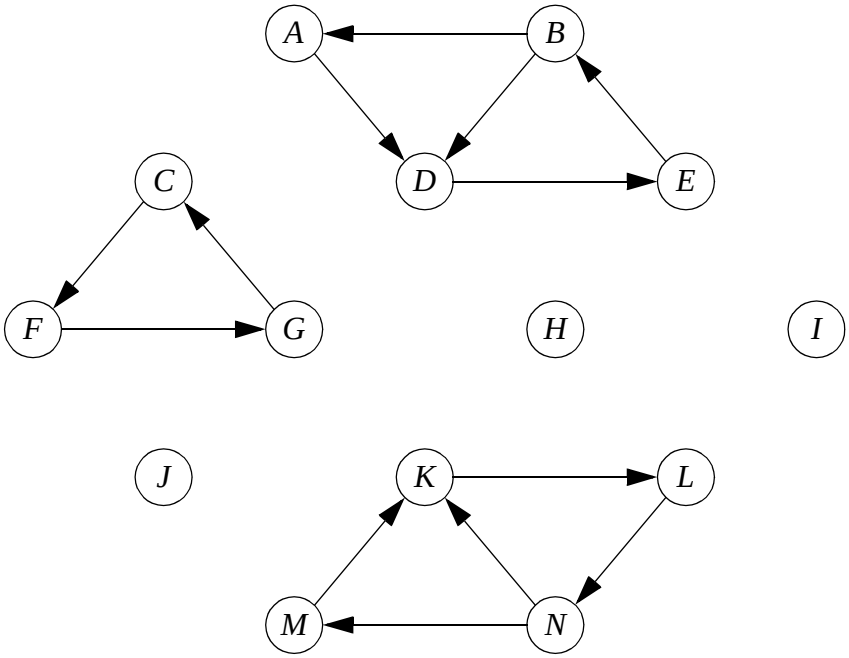
Anschließend wird der inverse Graph G_r erzeugt:



Die Depth-First-Spannbäume von G_r sind demnach:



Daraus ergeben sich die starken Komponenten von G:



Aufgabe 4

Die Sweep-Event-Struktur hat folgende Form:

((1, (1, 3, 1), *l*), (2, (2, 3, 6), *l*), (2, (2, 2, 7), *v*), (3,(2, 3, 6), *r*), (3, (3, 5, 3), *l*), (3, (1, 3, 1), *r*), (4, (4, 9, 4), *l*), (5, (5, 1, 5), *v*), (5, (3, 5,3), *r*), (8, (8, 1, 5), *v*), (9, (4, 9, 4), *r*))

Die Bezeichnungen *l*, *r* und *v* geben an, ob es sich um einen linken oder rechten Endpunkt eines horizontalen Segmentes oder um ein vertikales Segment handelt.

Die Funktion *isElem*(x, y_1, y_2) prüft, ob es in der Sweepline-Status-Struktur horizontale Segmente gibt, deren y-Werte im y-Intervall des übergebenen vertikalen Segmentes liegen und gibt dann die Paare y-Segment/x-Segment aus.

Ereignis	Aktion	Sweepline-Status-Struktur
(1, (1, 3, 1), *l*)	(1, 3, 1) einfügen	((1, 3, 1))
(2, (2, 3, 6), *l*)	(2, 3, 6) einfügen	((1, 3, 1), (2, 3, 6))
(2, (2, 2, 7), *v*)	*isElem*(2, 2, 7)	((1, 3, 1), (2, 3, 6))
(3, (2, 3, 6), *r*)	(2, 3, 6) entfernen	((1, 3, 1))
(3, (3, 5, 3), *l*)	(3, 5, 3) einfügen	((1, 3, 1), (3, 5, 3))
(3, (1, 3, 1), *r*)	(1, 3, 1) entfernen	((3, 5, 3))
(4, (4, 9, 4), *l*)	(4, 9, 4) einfügen	((3, 5, 3), (4, 9, 4))
(5, (5, 1, 5,), *v*)	*isElem*(5, 1, 5)	((3, 5, 3), (4, 9, 4))
(5, (3, 5, 3), *r*)	(3, 5, 3) entfernen	((4, 9, 4))
(8, (8, 1, 5), *v*)	*isElem*(8, 1, 5)	((4, 9, 4))
(9, (4, 9, 4), *r*)	(4, 9, 4) entfernen	

Aufgabe 5

Sei $i \in \{1, ..., n\}$ derjenige Parameter, der aussagt, dass i-1 Elemente bereits in $S[1], ..., S[i-1]$ sortiert vorliegen und dass noch n-i+1 Elemente in $S[i], ..., S[n]$ zu sortieren sind. In dieser Situation ist die Vorgehensweise wie folgt:

1. Wähle ein kleinstes Element aus dem Rest des Arrays S, d.h., aus $S[i], ..., S[n]$.
2. Vertausche das in Schritt 1 ausgewählte Element mit $S[i]$.
3. Sortiere den Rest des Arrays $S[i+1], ..., S[n]$.

Einen rekursiven Algorithmus kann man sich dann wie folgt vorstellen: Der Basisfall, in dem der Rekursionsabbruch eintritt, ist, dass $i = n$ gilt. In diesem Fall muss nur noch das letzte Element des Arrays sortiert werden. Da jedes einzelne Element bereits sortiert ist, brauchen wir nichts zu tun. Ist hingegen $i < n$, so finden wir das kleinste Element in $S[i], ..., S[n]$, vertauschen es mit $S[i]$ und sortieren $S[i+1], ..., S[n]$ rekursiv.

Eine rekursive Formulierung von *IterativeSelectionSort* kann dann zum Beispiel wie folgt aussehen:

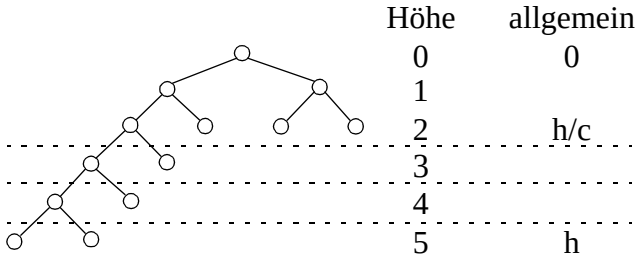
```
algorithm RecursiveSelectionSort(var S : RecArray, i, n : integer);
var j, temp: integer; min : keytype; minindex : 1..n;
begin
  if i < n then
     {Ist i = n, so ist dies der Basisfall, bei dem die Rekursion ohne Änderung des Arrays abgebrochen wird.}
     min := S[i].key;
     minindex := i;
     for j := i + 1 to n do
        if S[j].key < min then
           min := S[j].key;
           minindex := j
        fi
     od;
     temp := S[i];
     S[i] := S[minindex];
     S[minindex] := temp;
     RecursiveSelectionSort(S, i+1, n) { Sortiere den Rest des Arrays S.}
  fi
end RecursiveSelectionSort.
```

Ein Aufruf von *RecursiveSelectionSort*(*S*, 1, *n*) sortiert also das gesamte Array.

Aufgabe 6

(a)

Beginnen wir mit der Bestimmung der Untergrenze. Ein Baum enthält dann die minimale Anzahl möglicher Knoten, wenn die Blätter möglichst geringe Höhe aufweisen. Dies ist genau dann der Fall, wenn nur zwei Blätter auf der Höhe *h* liegen, während sich alle anderen Blätter möglichst nah an der Wurzel befinden, laut Voraussetzung also auf der Höhe *h*/*c*. Ein minimal gefüllter streng binärer Baum mit *h*=5 und *c*=5/2 sieht also z.B. folgendermaßen aus:



Ein solcher Baum besteht aus zwei Komponenten:

1. Ein vollständig gefüllter Baum der Höhe *h*/*c*.
2. Ein Teilstück, das aus dem vollständigen Baum herausragt. Es erstreckt sich über die restlichen *h* – *h*/*c* Ebenen und enthält in jeder Ebene 2 Knoten. Weniger als 2 Knoten pro Ebene kann es in einem streng binären Baum nicht geben, einmal abgesehen von der Wurzel in Ebene 0.

Die gesuchte minimale Anzahl $N_{\min}$ von Knoten in einem streng binären Baum ergibt sich dann als Summe der Knoten in beiden Komponenten:

$$N_{\min}(h,c) = N_2(h/c) + 2\,(h - h/c).$$

Hier bezeichnet $N_2(i)$ die Anzahl der Knoten eines vollständig gefüllten binären Baumes der Höhe i, also $N_2(i) = 2^{i+1} - 1$.

Die maximale Anzahl $N_{\max}$ von Knoten ergibt sich in einem Baum, wenn alle Blätter eine möglichst große Höhe aufweisen. Dies gilt für einen vollständigen Baum, in dem alle Blätter auf der Höhe h liegen. Da in unserem Fall jedoch $c > 1$ gilt, muß es ein Blatt b mit einer geringeren Höhe geben (nämlich h/c). Es fehlt also derjenige Teilbaum im vollständigen Baum, dessen Wurzel der Knoten b ist; b selbst ist allerdings noch vorhanden. Diese führt zu folgender Formel für $N_{\max}$:

$$N_{\max}(h,c) = N_2(h) - N_2(h - h/c) + 1.$$

(b)

Sei $N_k(h)$ die Anzahl der Knoten eines vollständig gefüllten Baumes der Höhe h, in dem jeder Knoten k Söhne hat: $N_k(h) = (k^{h+1} - 1)/(k-1)$. Dann gilt für die Ober- und Untergrenzen der Knotenanzahl:

$$N_{\min}(h, c) = N_k(h/c) + k\,(h - h/c)$$

bzw.

$$N_{\max}(h,c) = N_{\mathrm{k}}(h) - N_{\mathrm{k}}(h - h/c) + 1.$$