Halden (Heaps)

Algorithmen und Datenstrukturen

HOCHSCHULE AALEN
GRADUATE CAMPUS

LERNZIELE

- Kennenlernen einer neuen Datenstruktur „Heap“
- Verstehen der Eigenschaften von Heaps
- Algorithmen zum Herstellen dieser Eigenschaften
- Anwendungen von Heaps

EINFÜHRUNG

- Wir kennen (binäre) Heaps bereits (siehe Einbettung von binären Bäumen in Arrays)
- Also: Ein binärer Heap ist ein Arrayobjekt, das als vollständiger Baum angesehen werden kann
- Jeder Knoten des Baums entspricht einem Arrayelement. Dort werden die Werte des Knotens gespeichert.
- Der Baum ist vollständig in allen Ebenen bis auf evtl. die letzte Ebene.

3

NOTATION

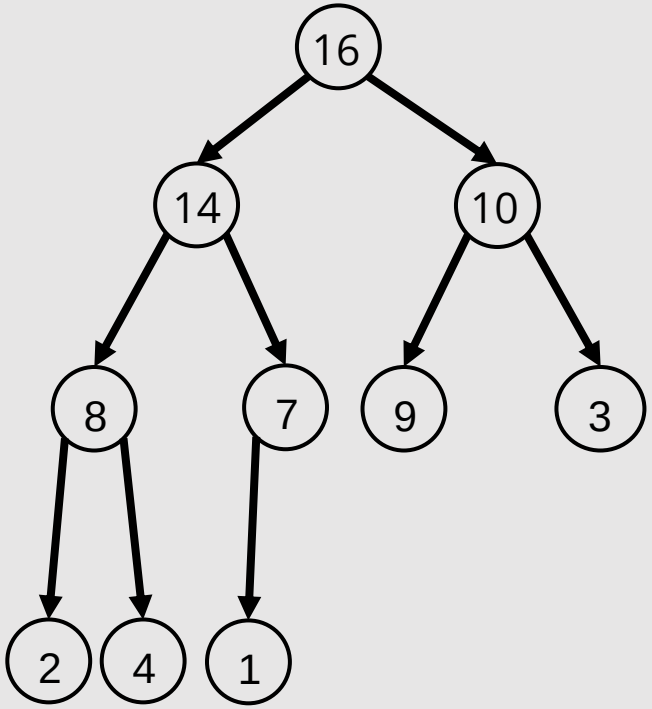
Ein Array *A*, der einen Heap repräsentiert, hat 2 Attribute:

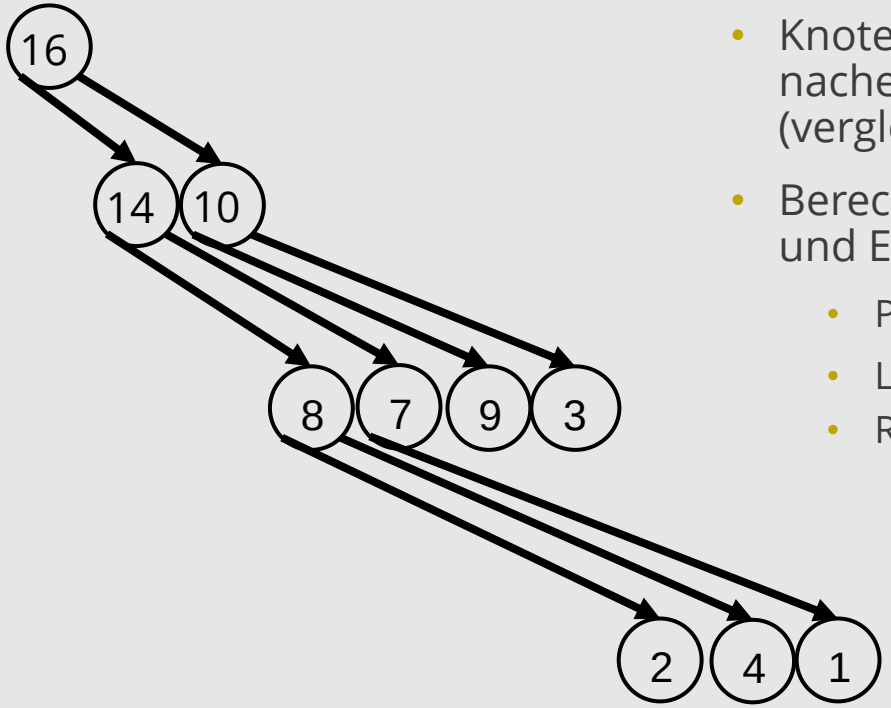
1. *A.length* = Anzahl der Elemente im Array
2. *A.size* = Anzahl der Elemente, die auch tatsächlich zum Heap gehören.

Gültige Elemente des Heaps sind nur die Arrayelemente von *A*[1] bis *A*[*A.size*] mit $A.size \leq A.length$

Die Wurzel des Baums ist *A*[1]

BEISPIEL





- Knoten werden zeilenweise nacheinander im Array abgelegt (vergleiche Level-Order-Treewalk)
- Berechnung der Position von Kind- und Elternknoten:
  - Parent(i) = $\left\lfloor \frac{i}{2} \right\rfloor$
  - Left(i) = 2 i
  - Right(i) = 2 i + 1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	14	10	8	7	9	3	2	4	1					

BERECHNUNG VON KIND- UND ELTERNKNOTEN

Parent(i)

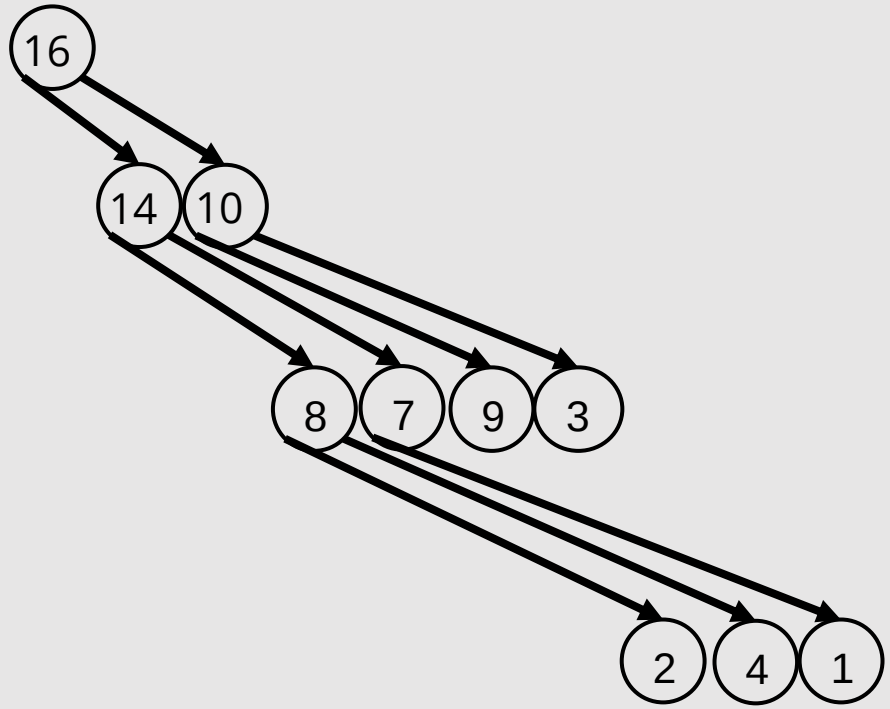
1. **function** PARENT(*i*)
2. **return** $\left\lfloor \frac{i}{2} \right\rfloor$
3. **end function**

Left(i)

1. **function** LEFT(*i*)
2. **return** 2*i*
3. **end function**

Right(i)

1. **function** RIGHT(*i*)
2. **return** 2*i* + 1
3. **end function**



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	14	10	8	7	9	3	2	4	1					

6

MIN-HEAP

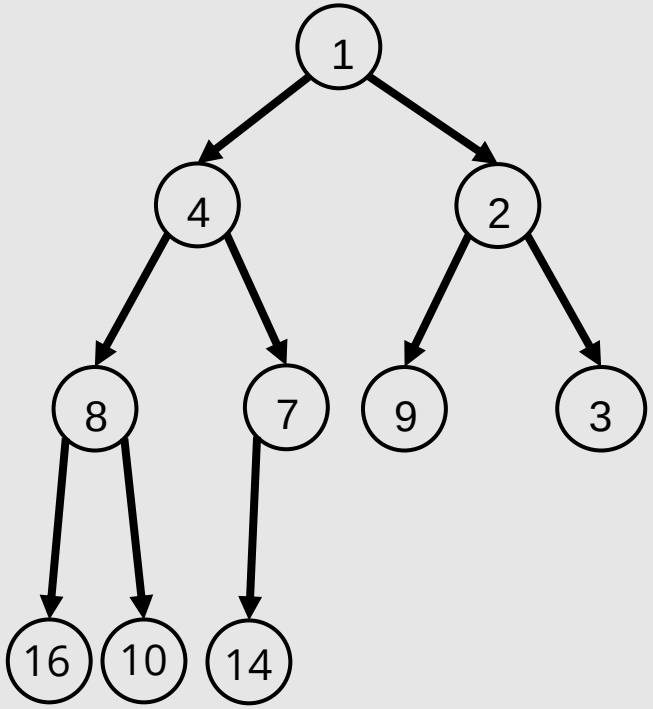
Definition

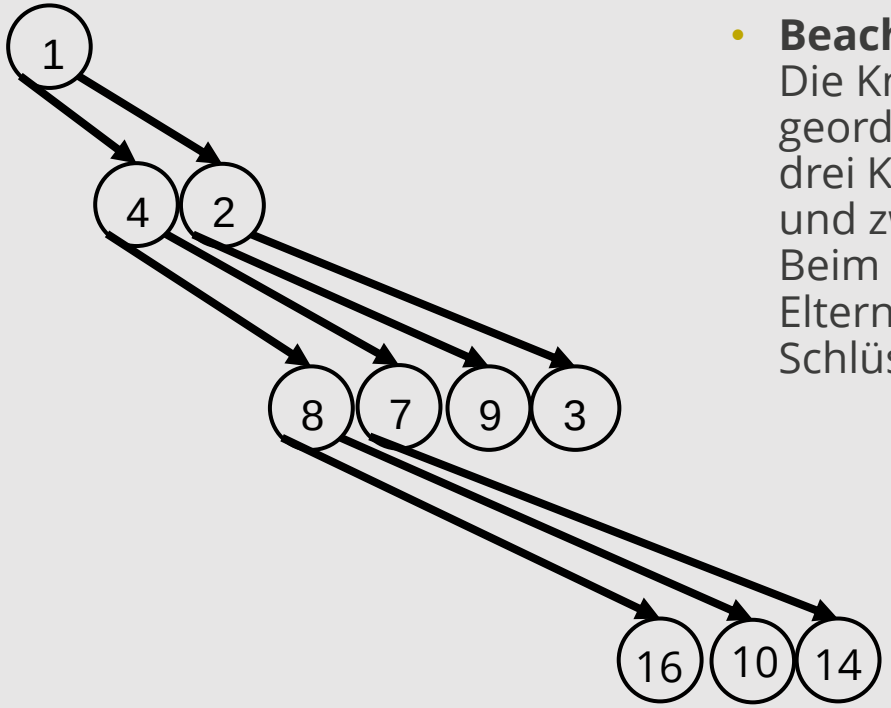
- Ein *Min-Heap ist ein Heap, für dessen Elemente gilt:*

$$\forall i : A[Parent(i)] \leq A[i]$$

- Die geforderte Eigenschaft heißt *Min-Heap-Eigenschaft*.
- Der Schlüssel eines Knotens ist mindestens gleich dem des Elternknotens.
- Die Wurzel ist das Element mit dem kleinsten Schlüssel.
- Alle Schlüssel in einem Teilbaum sind größer als der Schlüssel der Wurzel *n*.

BEISPIEL EINES MIN-HEAPS





- **Beachte:**
  Die Knoten sind nur partiell geordnet. Man betrachtet immer drei Knoten: Einen Elternknoten und zwei Kindknoten.
  Beim Min-Heap muss der Elternknoten den kleinsten Schlüsselwert enthalten.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	4	2	8	7	9	3	16	10	14					

8

MAX-HEAP

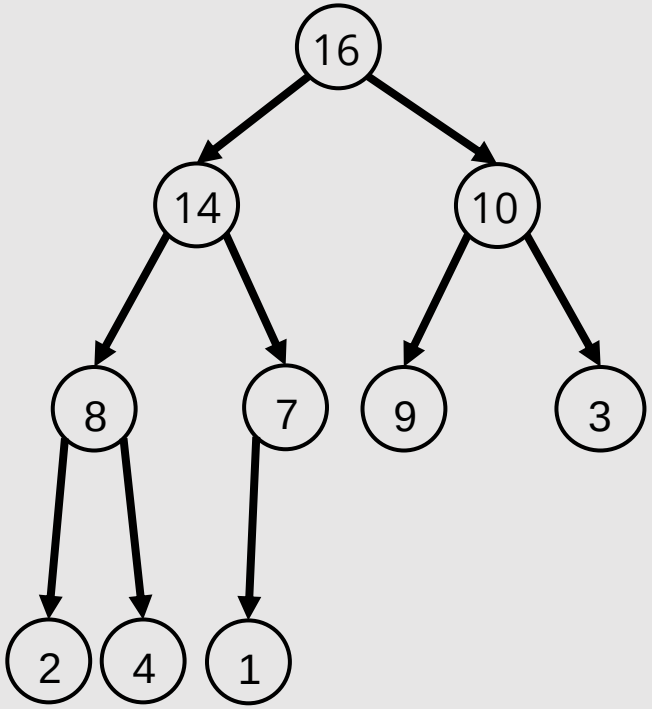
Definition

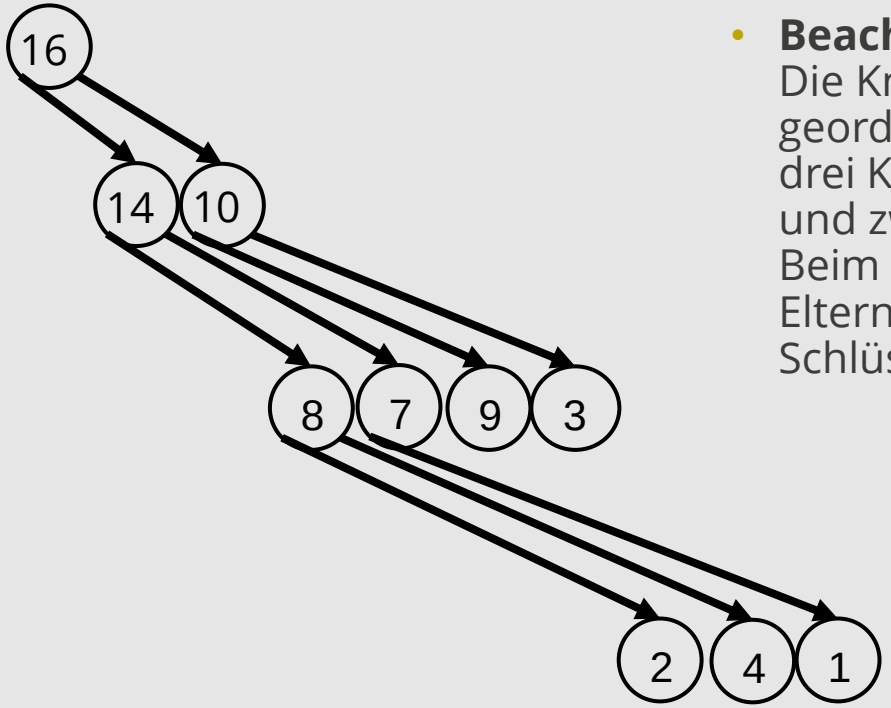
- Ein Max-Heap ist ein Heap, für dessen Elemente gilt:

$$\forall i : A[Parent(i)] \geq A[i]$$

- Die geforderte Eigenschaft heißt *Max-Heap-Eigenschaft*.
- Der Schlüssel eines Knotens ist maximal gleich dem des Elternknotens.
- Die Wurzel ist das Element mit dem größten Schlüssel.
- Alle Schlüssel in einem Teilbaum sind kleiner als der Schlüssel der Wurzel *n*.

BEISPIEL EINES MAX-HEAPS





- **Beachte:**
  Die Knoten sind nur partiell geordnet. Man betrachtet immer drei Knoten: Einen Elternknoten und zwei Kindknoten.
  Beim Max-Heap muss der Elternknoten den größten Schlüsselwert enthalten.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	14	10	8	7	9	3	2	4	1					

BEMERKUNGEN

- Für jeden Algorithmus muss angegeben werden, welche Art von Heaps er erwartet.
- Für den später zu besprechenden Heapsort-Algorithmus verwenden wir Max-Heaps.
- Min-Heaps werden typischerweise für Prioritätswarteschlagen (priority queues) verwendet.

BEMERKUNGEN

- Da man Heaps als Bäume interpretieren kann, können alle Begrifflichkeiten übernommen werden.
- Die *Höhe eines Knotens ist die Anzahl Kanten, über die der Knoten auf dem längsten einfachen Weg zu einem Blatt verbunden ist.*
- *Die Höhe h eines Heaps ist also* lg *n.*
- *Die Basisalgorithmen sind proportional zur Höhe h und haben damit eine Laufzeit*
- *O*(log *n*).

12

BEMERKUNGEN: ALGORITHMEN

Wir werden hier folgende Algorithmen entwickeln:

- MAX-HEAPIFY — Algorithmus zum Herstellen der Max-Heap-Eigenschaft ab einer bestimmten Position (Aufrechterhalten der Heap-Eigenschaft)
- BUILD-MAX-HEAP — Algorithmus zum Aufbau eines Max-Heaps aus einem unsortierten Array: (Erzeugen eines Heaps)

MAX-HEAPIFY

Max-Heapify(A, i)

A ist das Array (bzw. der Heap)
i ist der Index im Array

Max-Heapify soll die Max-Heap-Eigenschaft wieder herstellen, wenn sie verletzt ist:
Die Heap-Eigenschaft sei an der Stelle *i* verletzt.
Damit ist *A*[*i*] kleiner als eines seiner beiden Kinder.

Wir gehen davon aus, dass das linke Kind Left(i) und das rechte Kind Right(i) beide Wurzeln von gültigen Max-Heaps sind (für alle Knoten unter diesen Kindern gilt die Max-Heap-Eigenschaft).

Dann können wir A[*i*] hinunterwandern lassen, bis die Heap-Eigenschaft wieder hergestellt ist.

MAX-HEAPIFY

Max-Heapify(A, i)

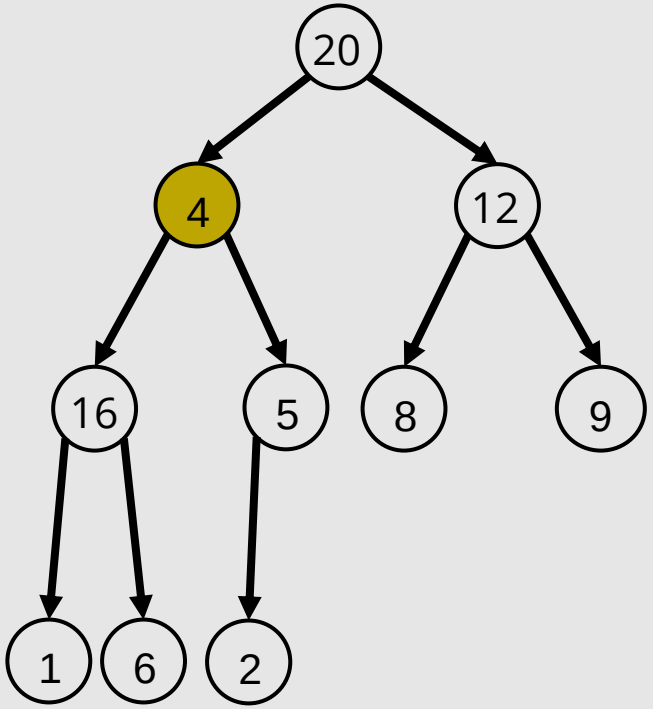
A ist das Array (bzw. der Heap)
i ist der Index im Array

Max-Heapify soll die Max-Heap-Eigenschaft wieder herstellen, wenn sie verletzt ist:
Die Heap-Eigenschaft sei an der Stelle *i* verletzt.
Damit ist *A*[*i*] kleiner als eines seiner beiden Kinder.

Wir gehen davon aus, dass das linke Kind Left(i) und das rechte Kind Right(i) beide Wurzeln von gültigen Max-Heaps sind (für alle Knoten unter diesen Kindern gilt die Max-Heap-Eigenschaft).

Dann können wir A[*i*] hinunterwandern lassen, bis die Heap-Eigenschaft wieder hergestellt ist.

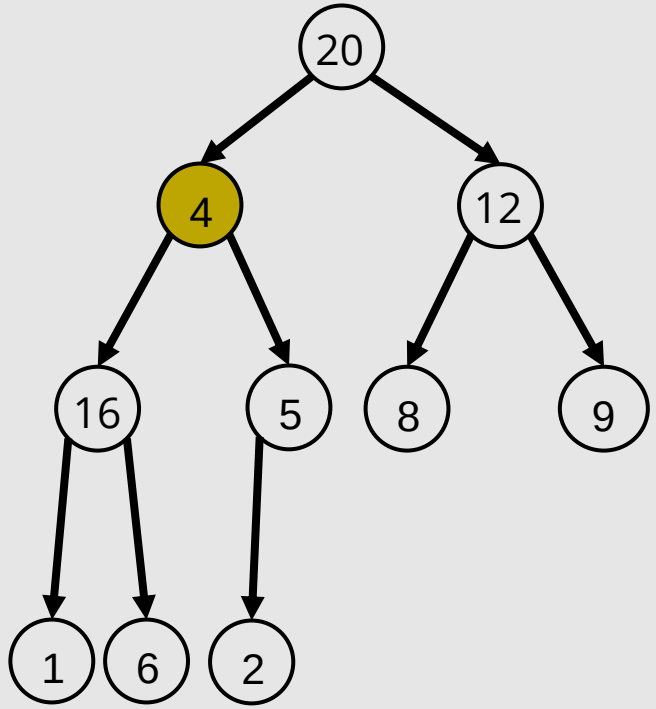
Beispiel: Die Heap-Eigenschaft ist in Index 2 verletzt, da die 4 sogar kleiner ist als beide darunterliegenden Zahlen 16 und 5 sind.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
20	**4**	12	16	5	8	9	1	6	2					

MAX-HEAPIFY ALGORITHMUS

```
1.  function MAX-HEAPIFY(A, i)
2.      left = LEFT(i)
3.      right = RIGHT(i)
4.      largest = i
5.      if left ≤ A.size and A[left] > A[i] then
6.          largest = left
7.      end if
8.      if right ≤ A.size and A[right] > A[largest] then
9.          largest = right
10.     end if
11.     if largest != i then
12.         Vertausche A[i] und A[largest]
13.         MAX-HEAPIFY(A, largest)
14.     end if
15. end function
```

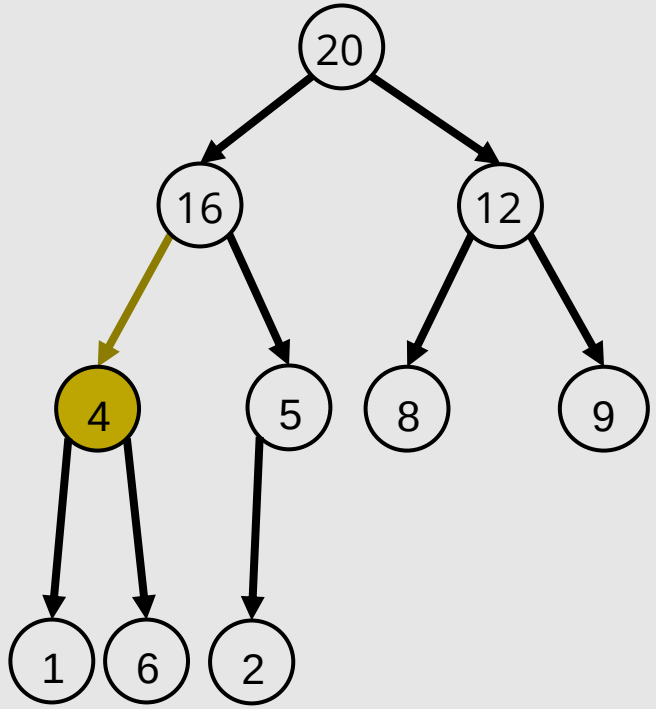


i=2
left = 4
right = 5

1	**2**	3	**4**	**5**	6	7	8	9	10	11	12	13	14	15
20	**4**	12	16	5	8	9	1	6	2					

MAX-HEAPIFY ALGORITHMUS

```
1.  function MAX-HEAPIFY(A, i)
2.      left = LEFT(i)
3.      right = RIGHT(i)
4.      largest = i
5.      if left ≤ A.size and A[left] > A[i] then
6.          largest = left
7.      end if
8.      if right ≤ A.size and A[right] > A[largest] then
9.          largest = right
10.     end if
11.     if largest != i then
12.         Vertausche A[i] und A[largest]
13.         MAX-HEAPIFY(A, largest)
14.     end if
15. end function
```

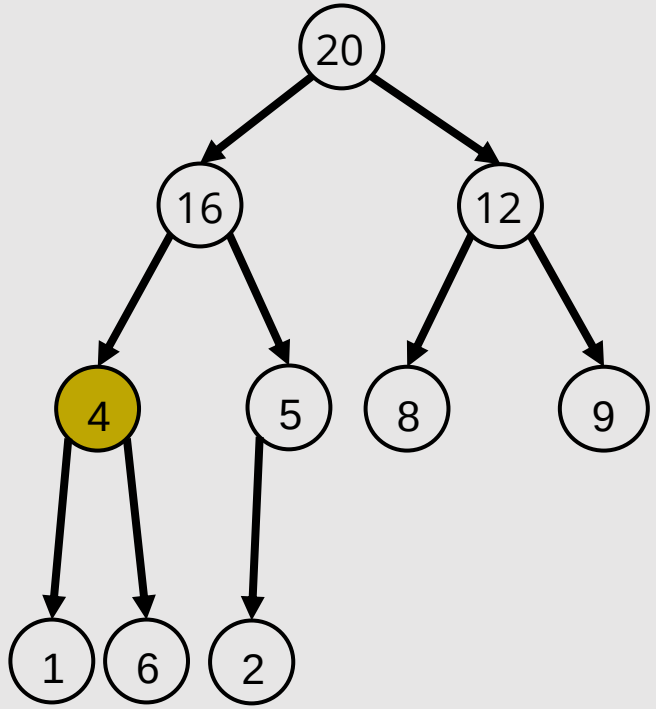


i=2
left = 4
right = 5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
20	**16**	12	**4**	5	8	9	1	6	2					

MAX-HEAPIFY ALGORITHMUS

```
1.  function MAX-HEAPIFY(A, i)
2.      left = LEFT(i)
3.      right = RIGHT(i)
4.      largest = i
5.      if left ≤ A.size and A[left] > A[i] then
6.          largest = left
7.      end if
8.      if right ≤ A.size and A[right] > A[largest] then
9.          largest = right
10.     end if
11.     if largest != i then
12.         Vertausche A[i] und A[largest]
13.         MAX-HEAPIFY(A, largest)
14.     end if
15. end function
```



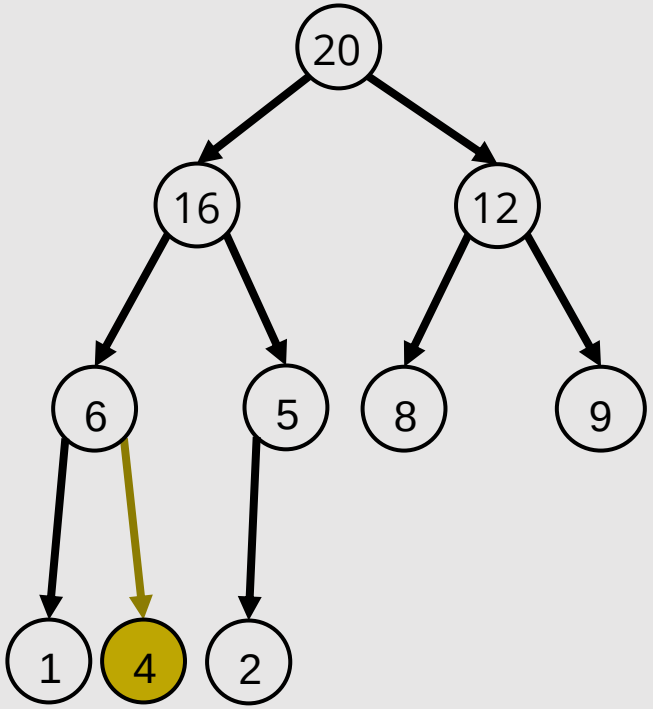
i=4
left = 8
right = 9

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
20	16	12	**4**	5	8	9	1	6	2					

18

MAX-HEAPIFY ALGORITHMUS

```
1.  function MAX-HEAPIFY(A, i)
2.      left = LEFT(i)
3.      right = RIGHT(i)
4.      largest = i
5.      if left ≤ A.size and A[left] > A[i] then
6.          largest = left
7.      end if
8.      if right ≤ A.size and A[right] > A[largest] then
9.          largest = right
10.     end if
11.     if largest != i then
12.         Vertausche A[i] und A[largest]
13.         MAX-HEAPIFY(A, largest)
14.     end if
15. end function
```

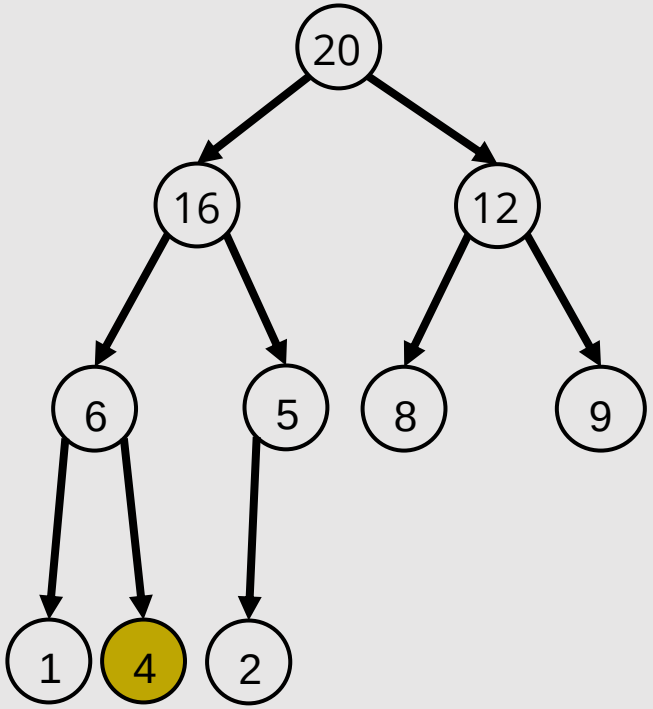


i=4
left = 8
right = 9

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
20	16	12	**6**	5	8	9	1	**4**	2					

MAX-HEAPIFY ALGORITHMUS

```
1.  function MAX-HEAPIFY(A, i)
2.      left = LEFT(i)
3.      right = RIGHT(i)
4.      largest = i
5.      if left ≤ A.size and A[left] > A[i] then
6.          largest = left
7.      end if
8.      if right ≤ A.size and A[right] > A[largest] then
9.          largest = right
10.     end if
11.     if largest != i then
12.         Vertausche A[i] und A[largest]
13.         MAX-HEAPIFY(A, largest)
14.     end if
15. end function
```



i=9
left = 18
right = 19

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
20	16	12	6	5	8	9	1	**4**	2					

MAX-HEAPIFY LAUFZEIT

Um MAX-HEAPIFY abzulaufen, müssen wir den aktuellen Teilbaum von der aktuellen Position nach unten laufen, und zwar nur in einem Pfad. Somit können wir die Laufzeit wieder wie bei der Suche nach Minimum und Maximum im Binärbaum abschätzen mit O(log n).

Mit dem sogenannten Master-Theorem lässt sich die Laufzeit von rekursiven Funktionen ebenfalls abschätzen. Das Ergebnis der Abschätzung mit dem Master-Theorem bringt das gleiche Ergebnis.

21

MASTERTHEOREM

Mit dem Mastertheorem lassen sich Laufzeiten von rekursiven Funktionen abschätzen, da diese nicht immer so leicht geschätzt werden können wie hier. Hierfür wird eine Rekursionsgleichung aufgestellt.

- Seien *a* und *b* Konstanten mit $a \geq 1$ und $b > 1$, $n \in \mathbb{N}$ und $f(n)$ eine von $T(n)$ unabhängige Funktion, dann ist die geschätzte Laufzeit $T(n)$ durch die Rekursionsgleichung $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ definiert.
  - a entspricht der Anzahl der Unterprobleme bzw. der Anzahl an rekursiven Aufrufen, die durchgeführt werden
  - n/b ist die Anzahl, durch die die ursprünglich zu bearbeitenden Elemente geteilt werden
  - f(n) ist die Arbeit, die in einer Rekursion durchgeführt werden muss. Die Laufzeit von f(n) ist die Zeit, die der eigentliche Algorithmus ohne rekursive Aufrufe benötigen würde.

- Dann unterscheidet das Mastertheorem für die Berechnung der Laufzeitabschätzung T(n) in Abhängigkeit von n drei Fälle.

- $$T(n) = \begin{cases} Fall\ 1: \Theta\left(n^{log_b a}\right) & wenn\ f(n) = O\left(n^{log_b a - \epsilon}\right) mit\ beliebigem\ \epsilon > 0 \\ Fall\ 2: \Theta(n^{log_b a} \cdot \log(n)) & wenn\ f(n) = \Theta\left(n^{log_b a}\right) \\ Fall\ 3:\ \Theta\left(f(n)\right) & wenn\ f(n) = \Omega\left(n^{log_b a + \epsilon}\right) mit\ beliebigem\ \epsilon > 0\ und \\ & a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n) für\ ein\ 0 < c < 1\ und\ große\ n \end{cases}$$

- Achtung: Das Mastertheorem lässt sich nicht für alle rekursiven Funktionen anwenden (speziell Fall 3 geht nur, wenn beide Bedingungen erfüllt sind!)

LAUFZEIT VON MAX-HEAPIFY

Laufzeit setzt sich zusammen aus:

- einem konstanten Anteil Θ(1) (Zeilen 2-11).
  Hier werden die Größenbeziehungen zwischen A[i] und LEFT(i) und RIGHT(i) festgestellt.

LAUFZEIT VON MAX-HEAPIFY (2)

- und einem von der Größe des Teilbaums abhängigen Teil. Hierfür gilt: Sei *n* die Größe des Teilbaums ab *i* .
- Dann haben die Kinder Teilbaumgrößen max. *2n/3.* Die Maximalzahl wird für Heaps erreicht, deren letzte Zeile exakt halb voll ist.

$$T(n) = a \cdot T\frac{n}{b} + f(n) = 1 \cdot T\frac{2n}{3} + f(n)$$

$$\text{mit } a = 1, b = \frac{3}{2}, \Rightarrow f(n) = \theta(n^{\overbrace{\log_{\frac{3}{2}}(1)}^{0}}) = \theta(1) \text{ (Fall 2)}$$

LAUFZEIT VON MAX-HEAP

Insgesamt ist damit

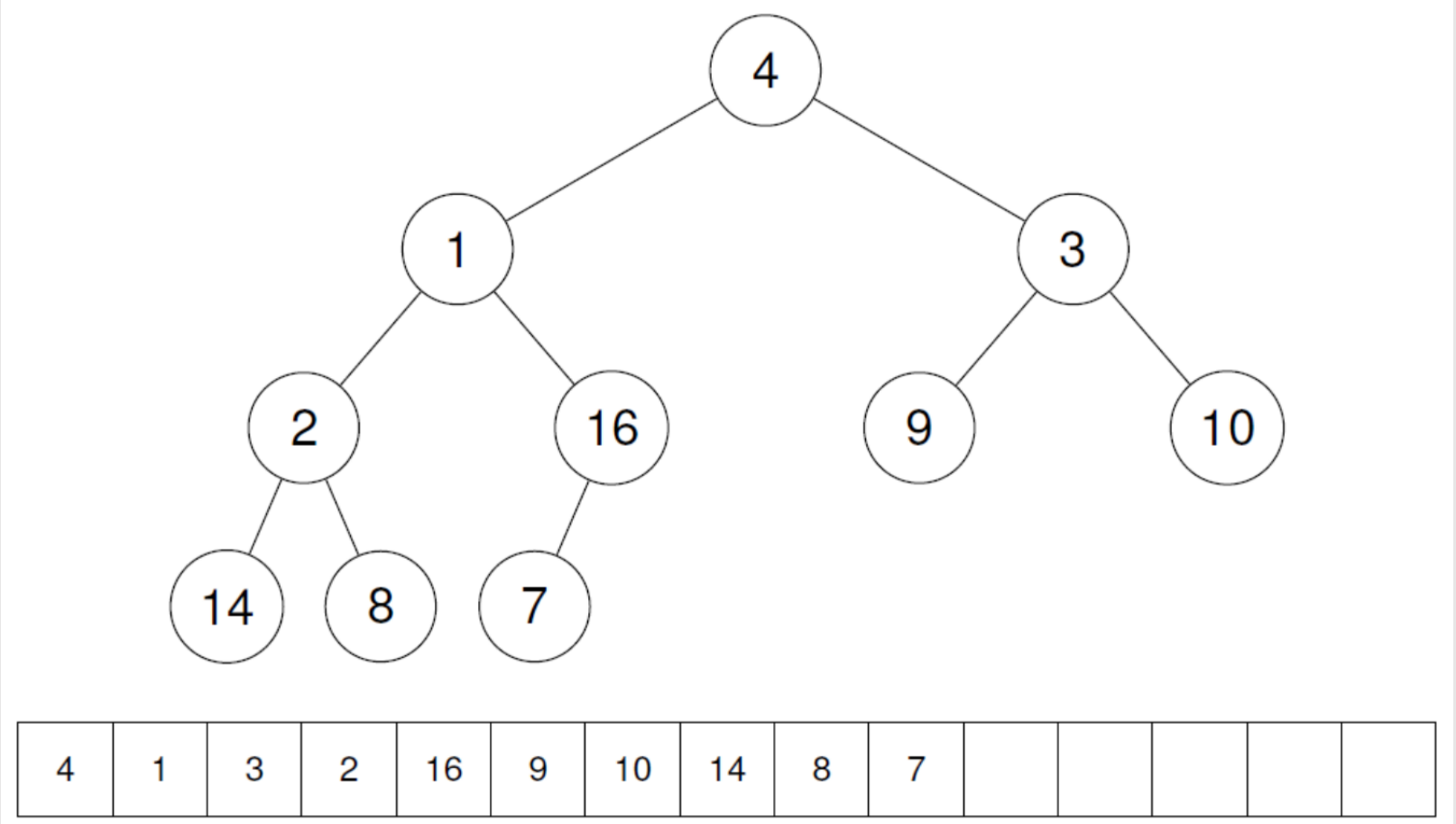
$$T(n) = \theta(n^{\log_b a} \cdot logn) = \theta(n^0 \cdot logn) = \mathrm{O}(logn)$$

- Intuitiv war eigentlich klar, dass T(n) = O(h).

BUILD-MAX-HEAP (A)

- Benutze *Max-Heapify*, um einen Heap von den Blättern her zu erzeugen.
- Die Blätter haben Indizes $\left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n$.
- Jedes Blatt stellt einen trivialen Max-Heap dar.

AUSGANGSSITUATION

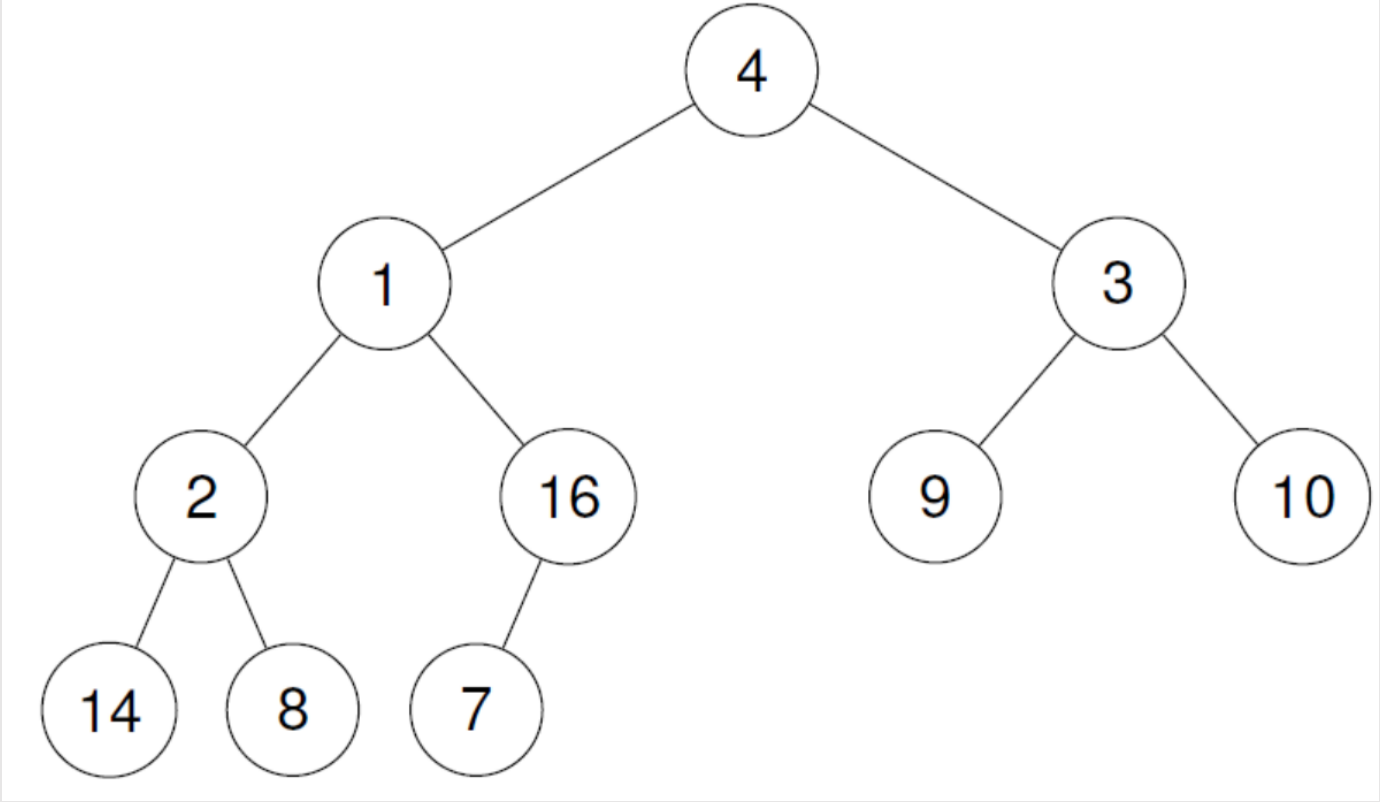


BUILD-MAX-HEAP

```
1.  function BUILD-MAX-HEAP(A)
2.      for i = A.size/2 downto 1 do
3.          MAX-HEAPIFY(A, i)
4.      end for
5.  end function
```

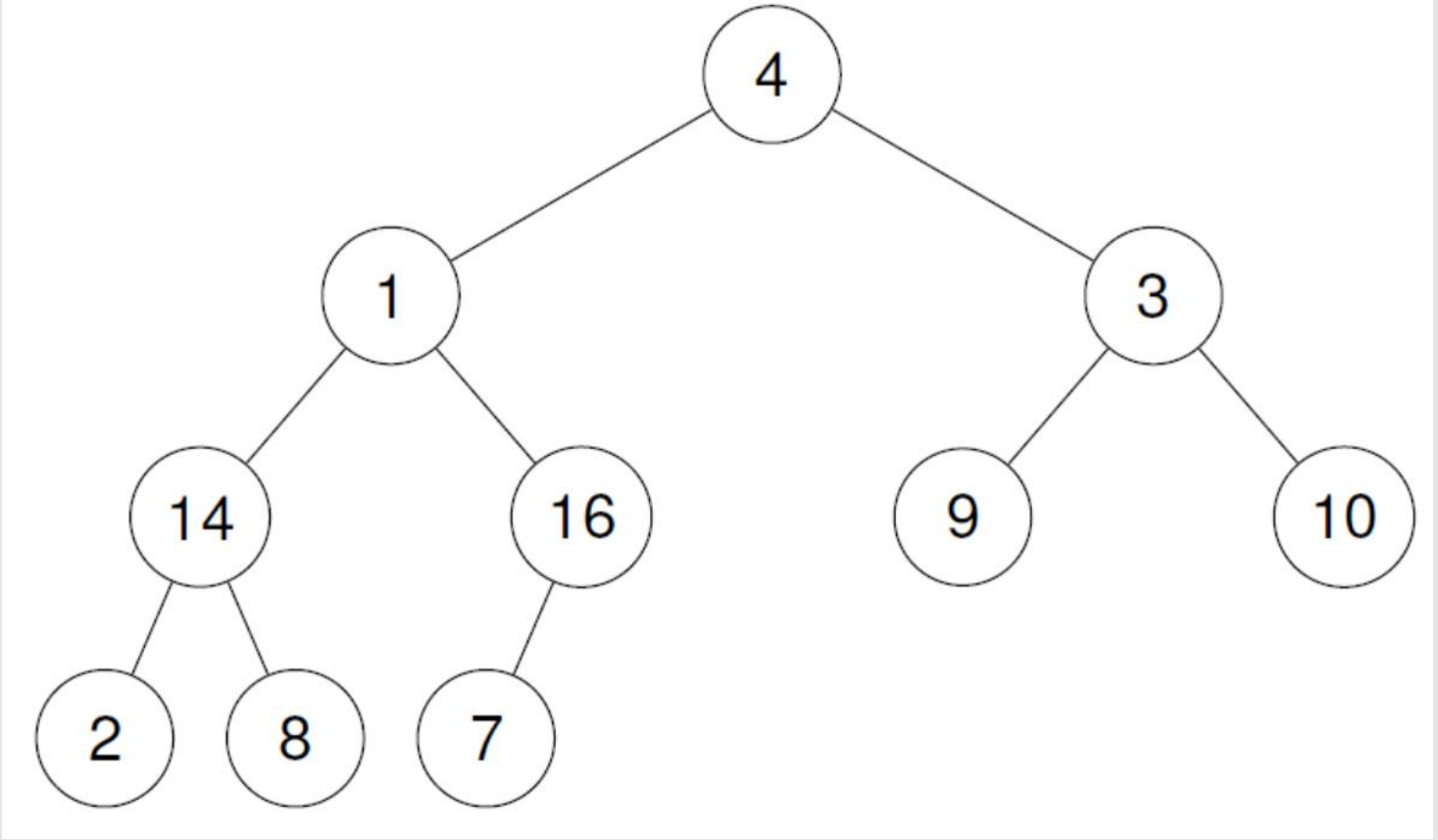
BUILD-MAX-HEAP

Übung: Stellen Sie die Max-Heap-Eigenschaft unter Verwendung von Build-Max-Heap her.



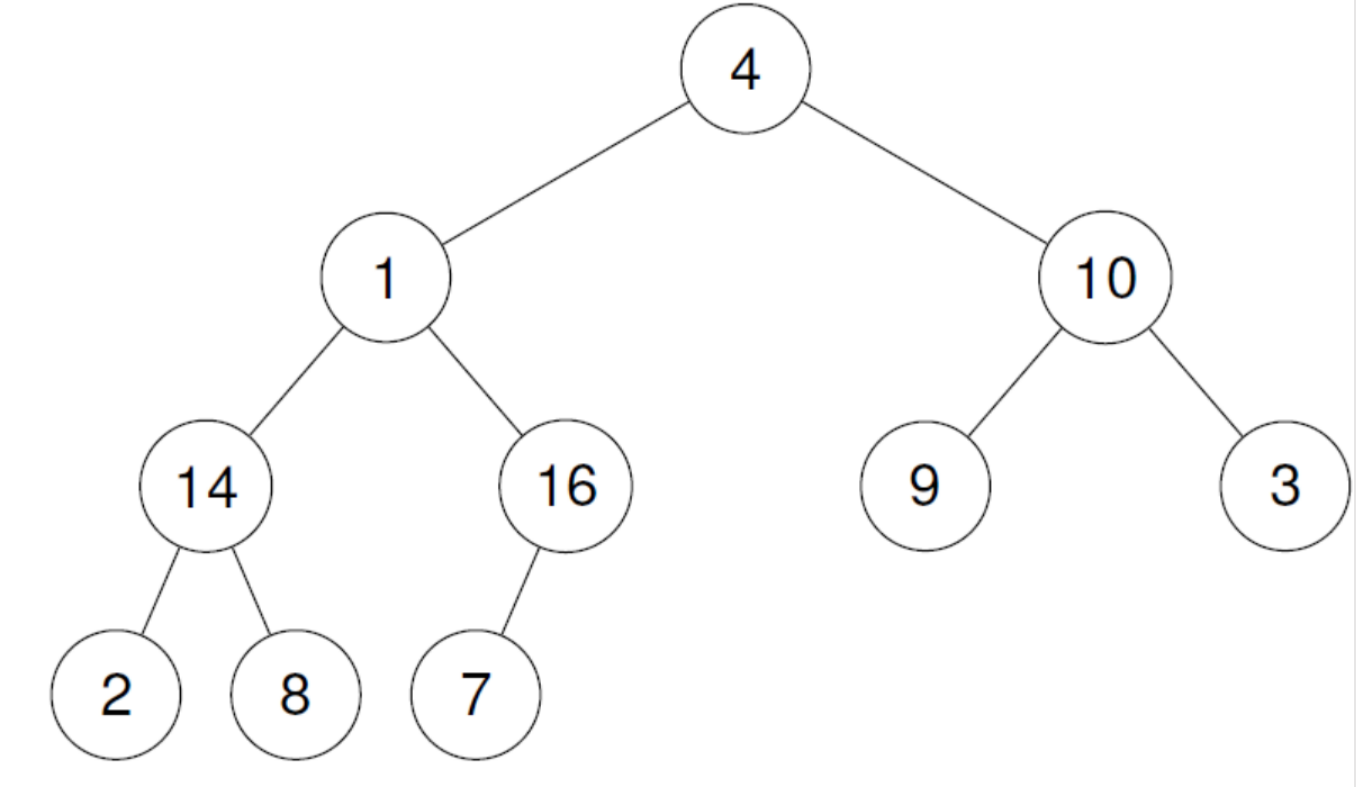
BUILD-MAX-HEAP

- Lösung zur Übung:



BUILD-MAX-HEAP

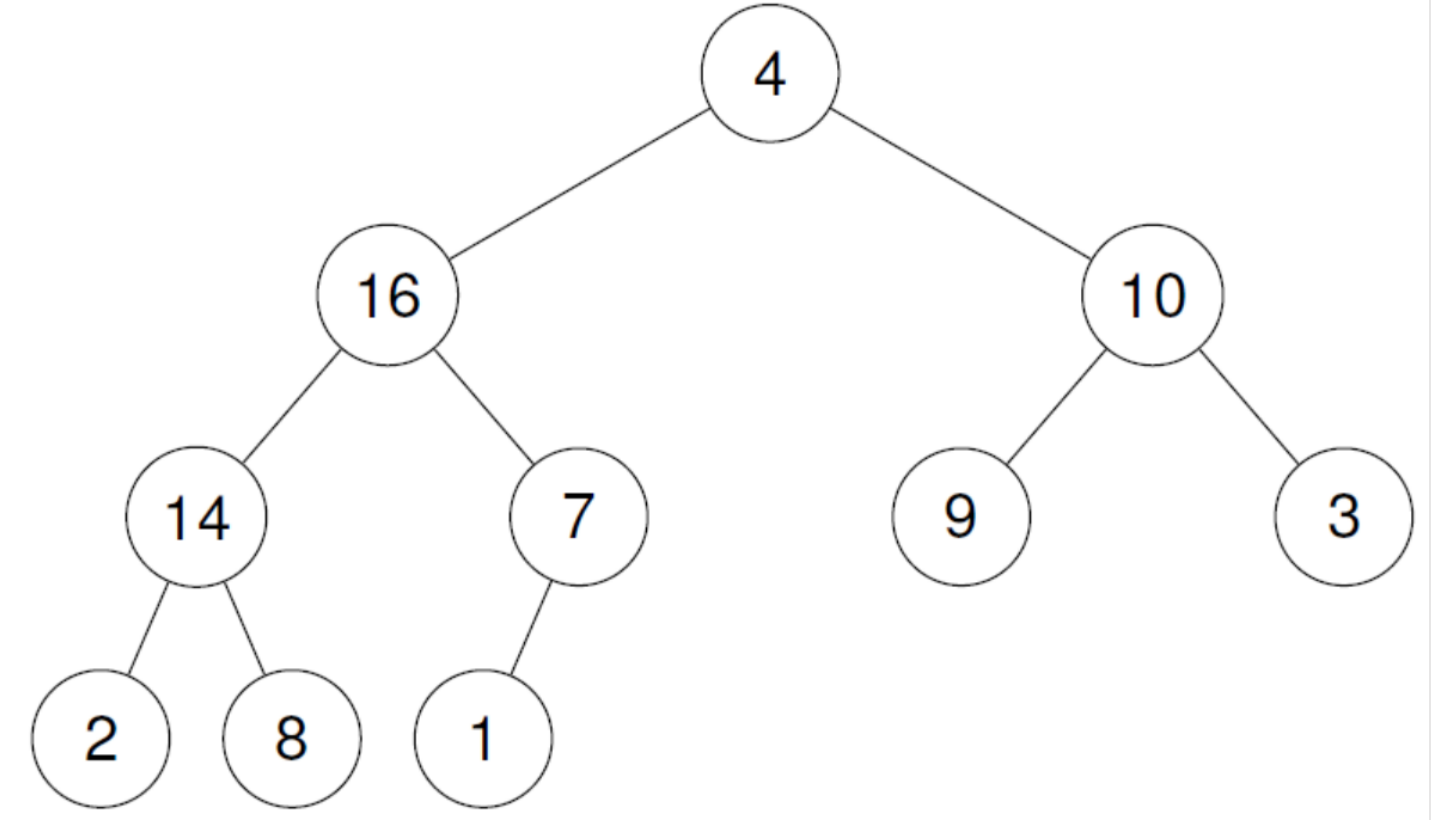
Lösung zur Übung:



31

BUILD-MAX-HEAP

- Lösung zur Übung:



32

BUILD-MAX-HEAP: KORREKTHEIT

Die Korrektheit von Build-Max-Heap kann mit der Vollständigen Induktion bewiesen werden.
Hierfür definieren wir zuerst eine Aussage, die gelten soll (Schleifeninvariante) und prüfen anschließend, ob sie vor dem ersten Betreten der Schleife (Initialisierung), nach jedem Schleifendurchlauf (Aufrechterhaltung) und nach Beenden der Schleife (Terminierung) gilt.

Schleifeninvariante

- Beim Start jeder Schleife (Zeilen 3-5) sind die Knoten $i + 1, i + 2, \ldots n$ Wurzeln eines Max-Heaps.

1.) Initialisierung

$$i = \left\lfloor \frac{n}{2} \right\rfloor$$

Alle Knoten $i = \left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n$ sind Blattknoten und damit trivialerweise Wurzeln von Max-Heaps.

33

BUILD-MAX-HEAP: KORREKTHEIT

2.) Aufrechterhaltung

Wir prüfen zuerst, ob die Voraussetzung für MAX-HEAPIFY erfüllt ist, dass die Kindknoten Wurzeln von Teilbäumen sind, die selbst Max-Heaps sind (vgl. Max-Heap-Algorithmus)

- Die Kinder von *i* haben Indizes > *i, nämlich 2i und 2i+1*.
- Nach der Schleifeninvariante haben alle Knoten mit Indizes > *i bereits die* Max-Heap-Eigenschaft, insbesondere also auch 2*i* und 2*i* + 1.

⇒ Voraussetzung für MAX-HEAPIFY ist erfüllt.

- MAX-HEAPIFY erhält die Max-Heap-Eigenschaft für alle Knoten *i* + 1, . . .*n* und stellt sie für den Knoten mit Index *i* her.
- Nachdem *i* vor dem nächsten Schleifendurchlauf um eins verringert (dekrementiert) wird, ist zur nächsten Iteration die Schleifeninvariante wieder erfüllt, da wieder alle Knoten mit Indizes > i bereits die Max-Heap-Eigenschaft besitzen.

34

BUILD-MAX-HEAP: KORREKTHEIT

3) Terminierung:

- Am Ende der Schleife wird i noch einmal verringert und ist dann 0, da die Schleife nur bis i=1 betreten wird.
- Alle Knoten von 1 bis n haben nun die Max-Heap-Eigenschaft.

Ergebnis:

Mit der Vollständigen Induktion konnten wir somit bewiesen, dass der Algorithmus einen vollständigen Max-Heap erzeugt und somit korrekt ist.

LAUFZEIT VON BUILD-MAX-HEAP

Einfache Abschätzung der oberen Grenze:

- *Max-Heapify* hat $O(\log n)$.
- es gibt $O(n)$ Aufrufe.
- Laufzeit ist also $T(n) = O(n \cdot \log n)$.

Diese Grenze ist korrekt, aber nicht so eng wie möglich.

LAUFZEIT VON BUILD-MAX-HEAP

Exakte Abschätzung

- Der Aufwand beim Aufruf von MAX-HEAPIFY ist abhängig von der Höhe des Teilbaums ab diesem Knoten. Wir benötigen daher eine genauere Abschätzung des Aufwands für die einzelnen Ebenen.
- Ein balancierter Binärbaum mit Kapazität *n* hat Höhe *h* = log2 *n*
- Es gibt höchstens $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ Knoten mit (Teilbaum-)Höhe *h* in einem *n*-elementigen Heap. Dabei ist die Höhe auf der letzten Ebene 0 usw.
- Der Aufruf von MAX-HEAPIFY auf einem Teilbaum mit Höhe *h* hat *O*(*h*).

37

LAUFZEIT VON BUILD-MAX-HEAP

Übung zur Exakten Abschätzung

- Zeichnen Sie einen Baum mit Kapazität $n = 15$
- Wie groß ist seine Höhe?
-
- Betrachten Sie alle Ebenen ab der untersten Ebene und berechnen Sie die Anzahl der Knoten, die eine Höhe haben, die der Höhe von Teilbäumen in dieser Ebene entspricht.

- Vergleichen Sie mit $\left\lceil \frac{n}{2^{h+1}} \right\rceil$

38

LAUFZEIT VON BUILD-MAX-HEAP

Exakte Abschätzung

- Es gilt:

$$n = 15 \Rightarrow h = \log_2 15 = 3{,}91$$

h	$\left\lceil \frac{15}{2^{h+1}} \right\rceil$
0	8
1	4
2	2
3	1

LAUFZEIT

Exakte Abschätzung

- Gesamtkosten

$$\sum_{h=0}^{\lfloor log\ n \rfloor} \frac{n}{2^{h+1}} \cdot O(h) = O(n \cdot \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^{h+1}}) = O(\frac{n}{2} \cdot \sum_{h=0}^{\lfloor log\ n \rfloor} \frac{h}{2^{h}})$$

40

LAUFZEIT

Hilfssatz

Für |x| < 1 gilt:

$$\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2}$$

Hier: $x = \frac{1}{2}$

$$\sum_{k=0}^{\infty} \frac{k}{2^k} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$$

EXAKTE ABSCHÄTZUNG

Gesamtkosten sind also

$$O(\frac{n}{2} \cdot \sum_{h=0}^{\lfloor ld\ n \rfloor} \frac{h}{2^h}) = O(n)$$

- Die Funktion erzeugt einen Max-Heap also in linearer Zeit.
- Alle Überlegungen gelten gleichermaßen für Min-Heaps.

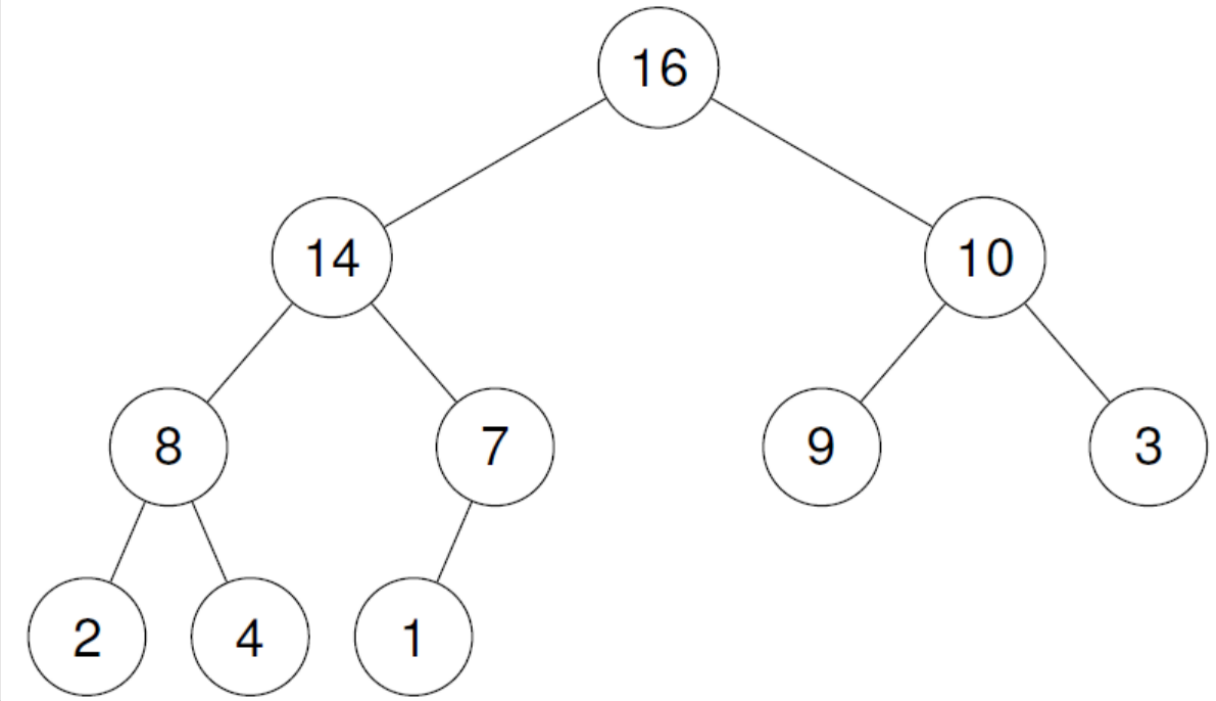
SORTIEREN MIT HEAPS

Ziel

- Entwicklung eines Sortierverfahrens unter Benutzung von Max-Heaps
- Lohnt sich, da BUILD-MAX-HEAP() $T(n) = O(n)$ hat
- Das zu sortierende Array A muss zu Beginn gültiger Max-Heap sein. Das können wir durch den Aufruf von BUILD-MAX-HEAP() vor dem Sortieren sicherstellen

HEAPSORT

- Ausgangssituation



- Was kann man beobachten?

HEAPSORT ALGORITHMUS

Vorgehen

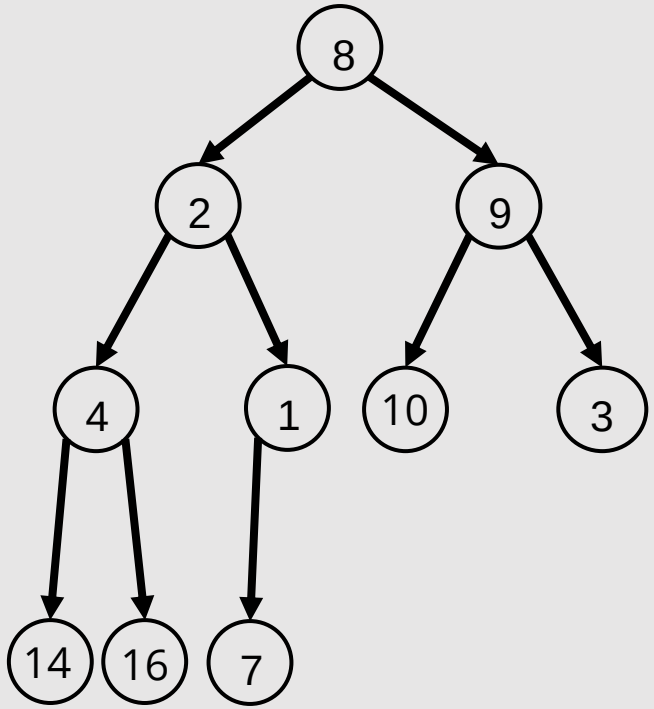
- *A* muss zu Beginn ein gültiger Max-Heap sein.
- Damit steht das größte Element an Position *A*[1].
- Das größte Element soll aber bei aufsteigender Sortierung das letzte Element sein. A[1] muss also mit A[n] vertauscht werden.
- A[n] liegt nun bereits an der richtigen Stelle, d.h. der Algorithmus muss nur noch die Elemente A[1] bis A[n-1] sortieren.
- Um das wieder mit der Max-Heap-Eigenschaft tun zu können, muss diese für A[1] bis A[n-1] wiederhergestellt werden (mit MAX-HEAPIFY).
- Der Vorgang kann nun so lange wiederholt werden, bis nur noch ein Element übrig ist, welches automatisch einen Max-Heap darstellt. Dann sind alle Elemente sortiert.

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

45

HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**



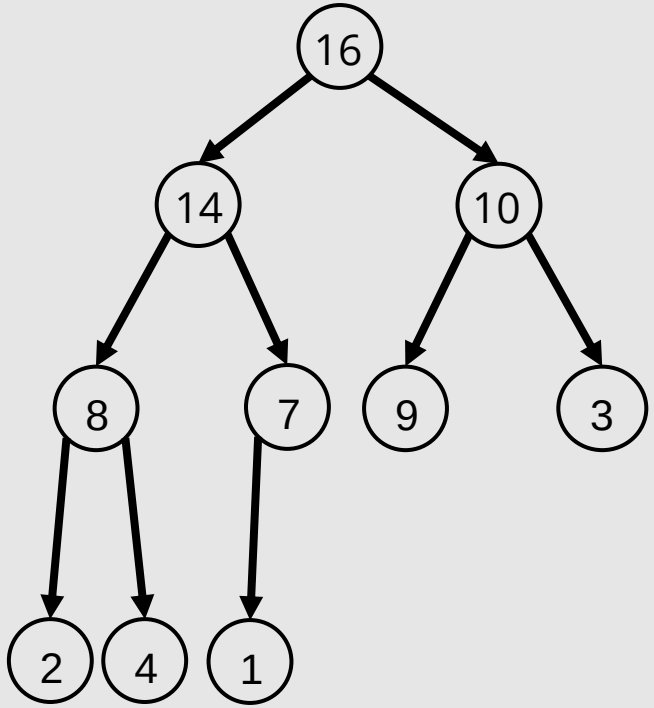
1	2	3	4	5	6	7	8	9	10
8	2	9	4	1	10	3	14	16	7

A.size

46

HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. **BUILD-MAX-HEAP(A)**
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**



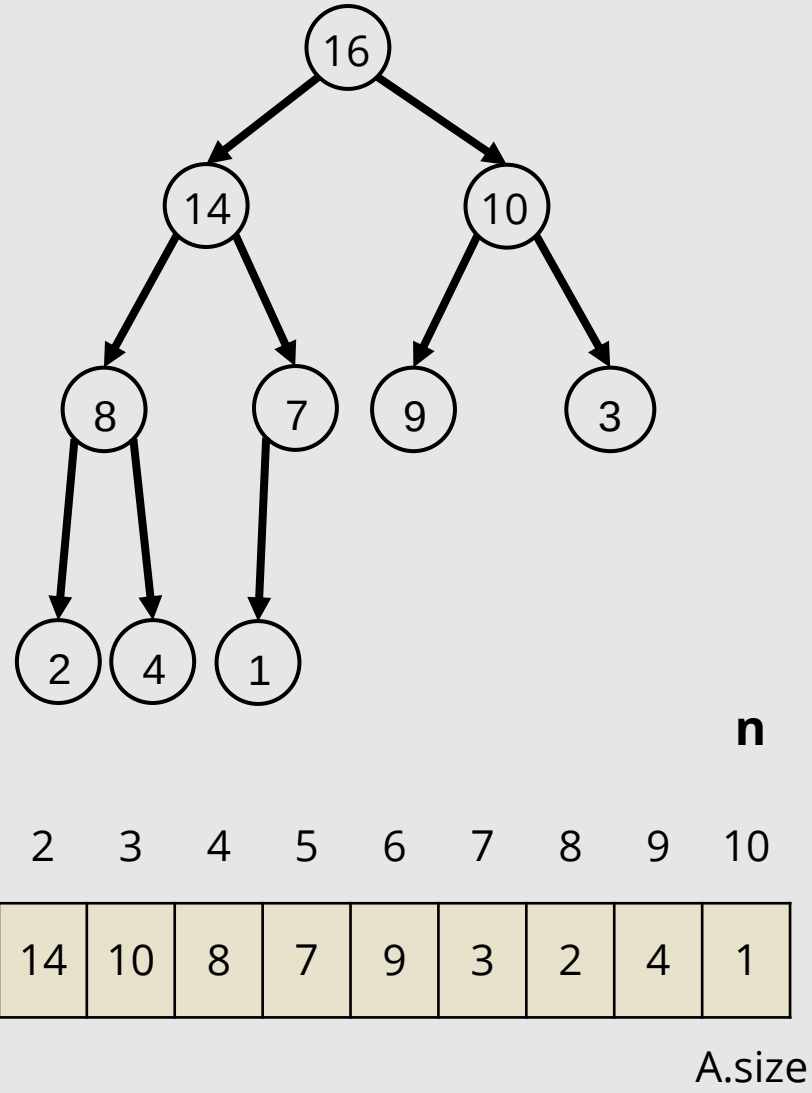
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

A.size

47

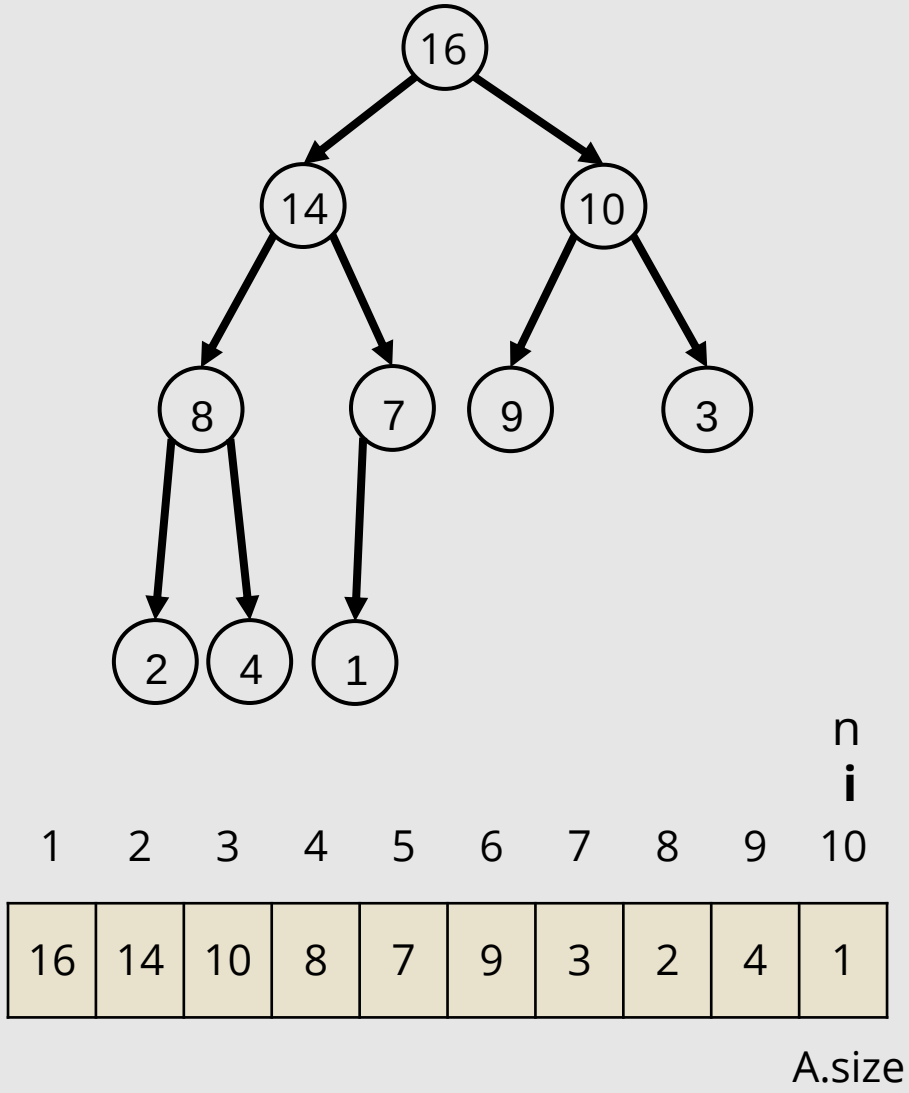
HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. ***n = A.size***
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**



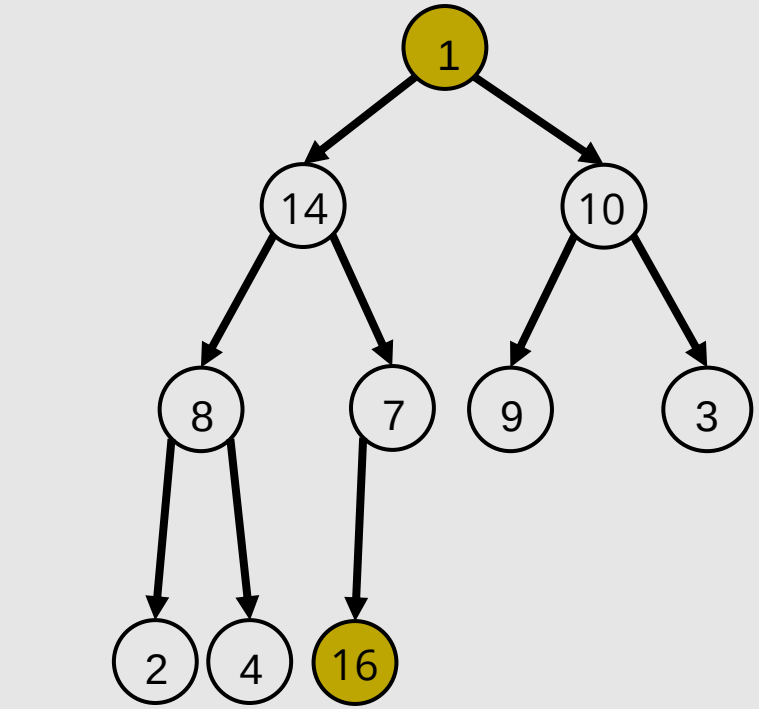
HEAPSORT BEISPIEL

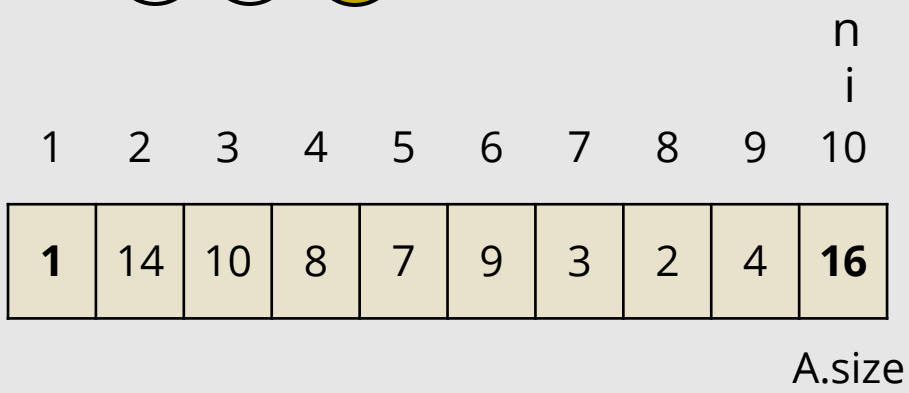
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** ***i*** **=** ***n*** **downto 2 do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**



HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. **vertausche** ***A*[1] ↔ *A*[*i*]**
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

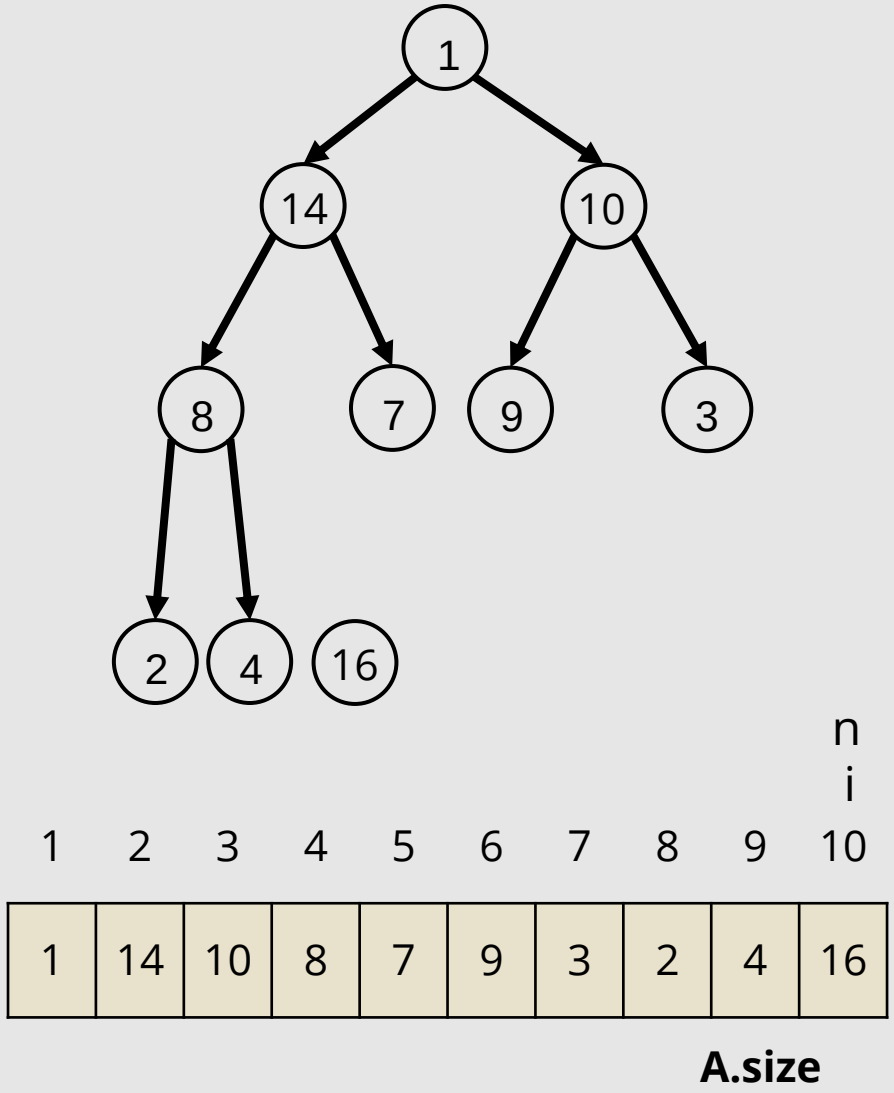




50

HEAPSORT BEISPIEL

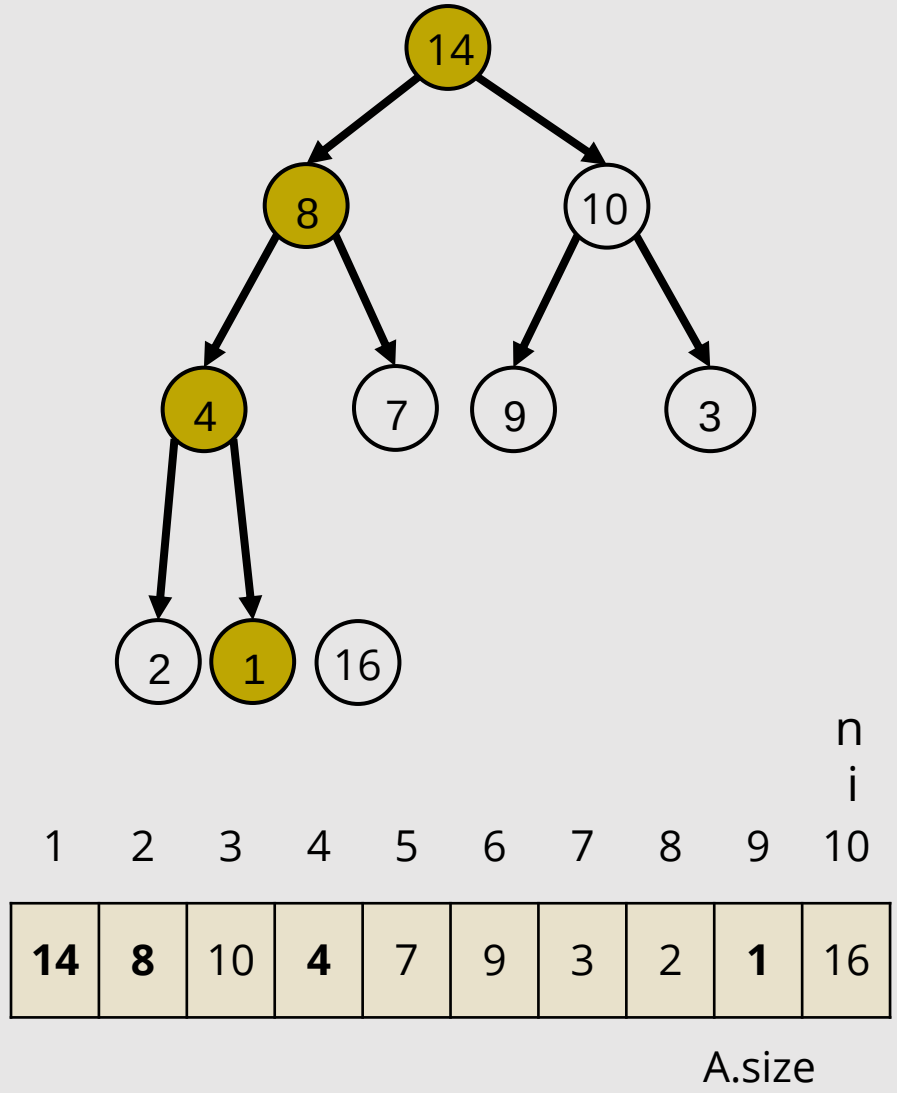
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. ***A.size* = *A.size* − 1**
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**



51

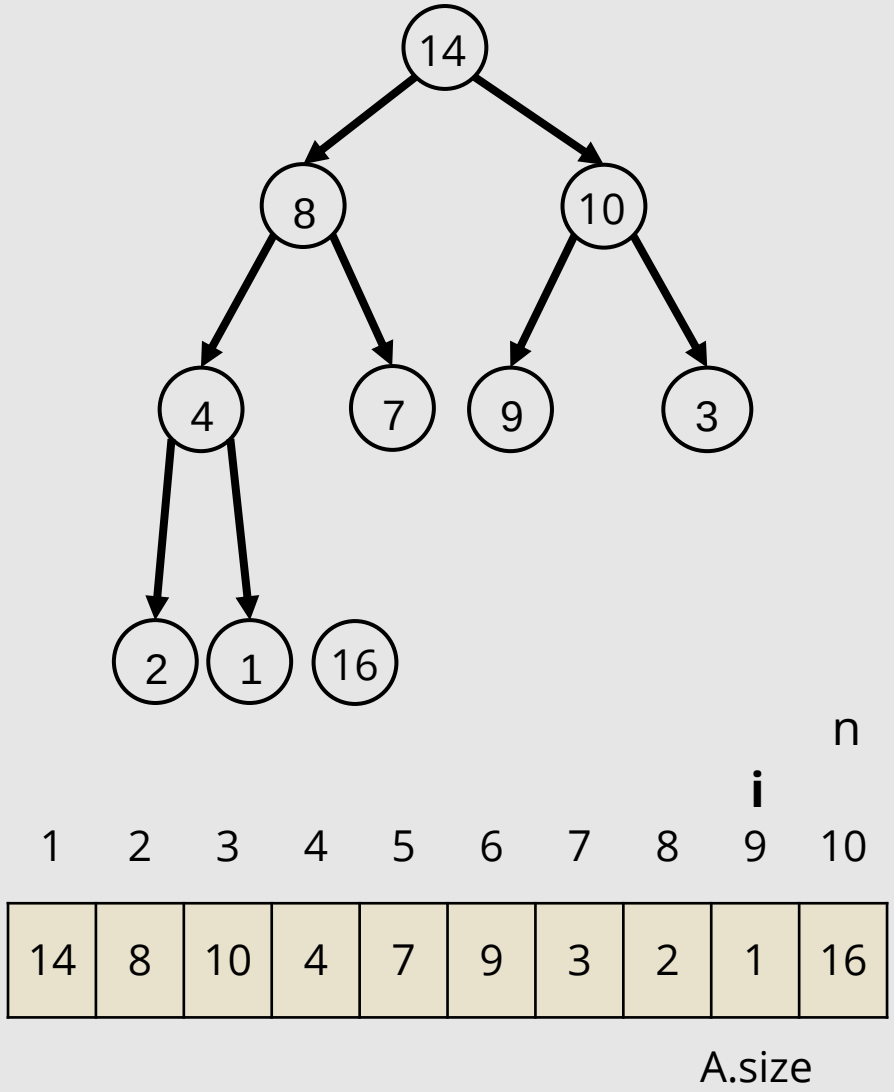
HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. **MAX-HEAPIFY(A, 1)**
8. **end for**
9. **end function**



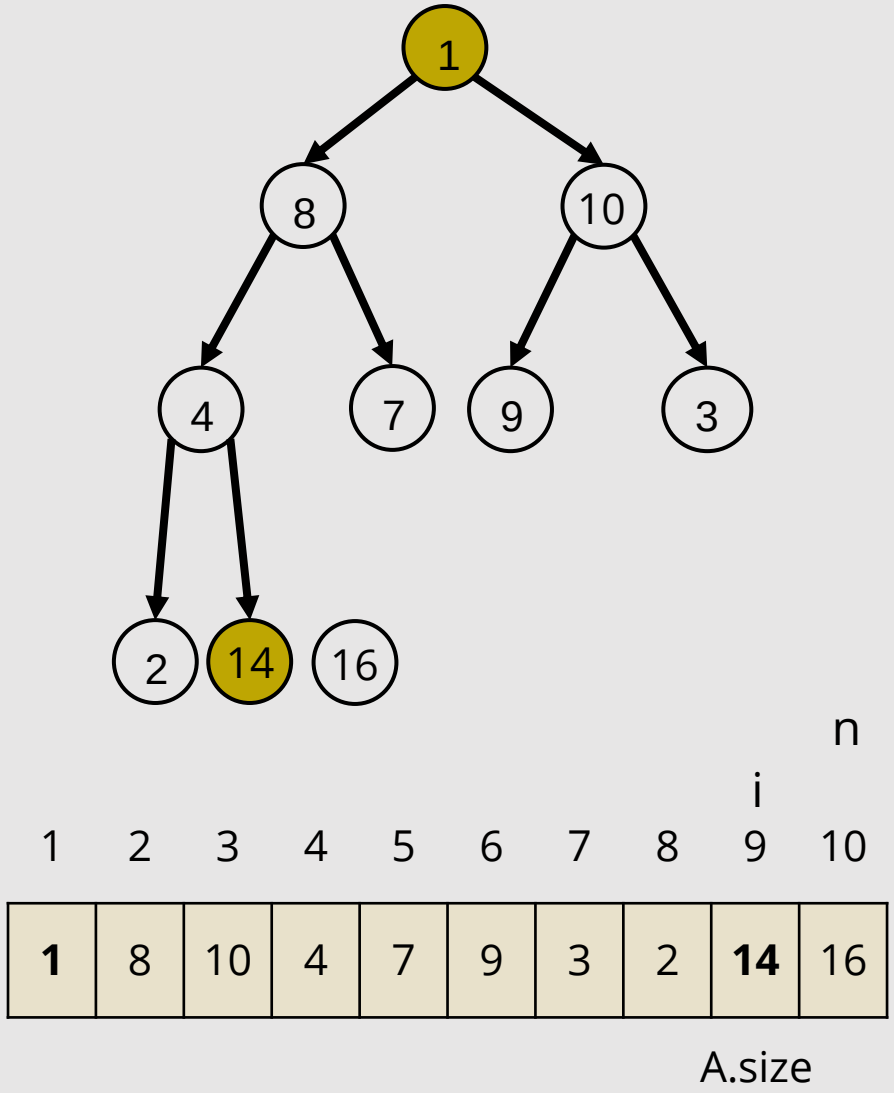
HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** ***i*** **=** ***n*** **downto 2 do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**



HEAPSORT BEISPIEL

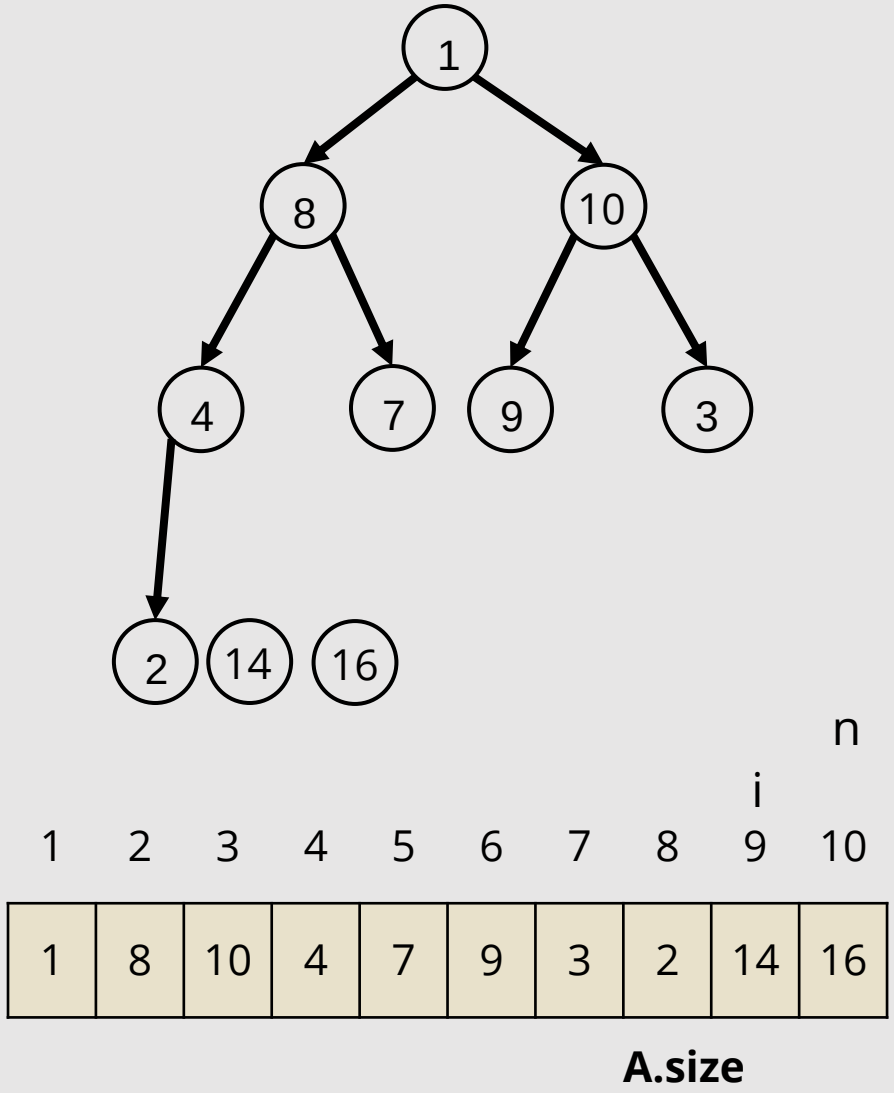
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. **vertausche** ***A*[1] ↔ *A*[*i*]**
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**



54

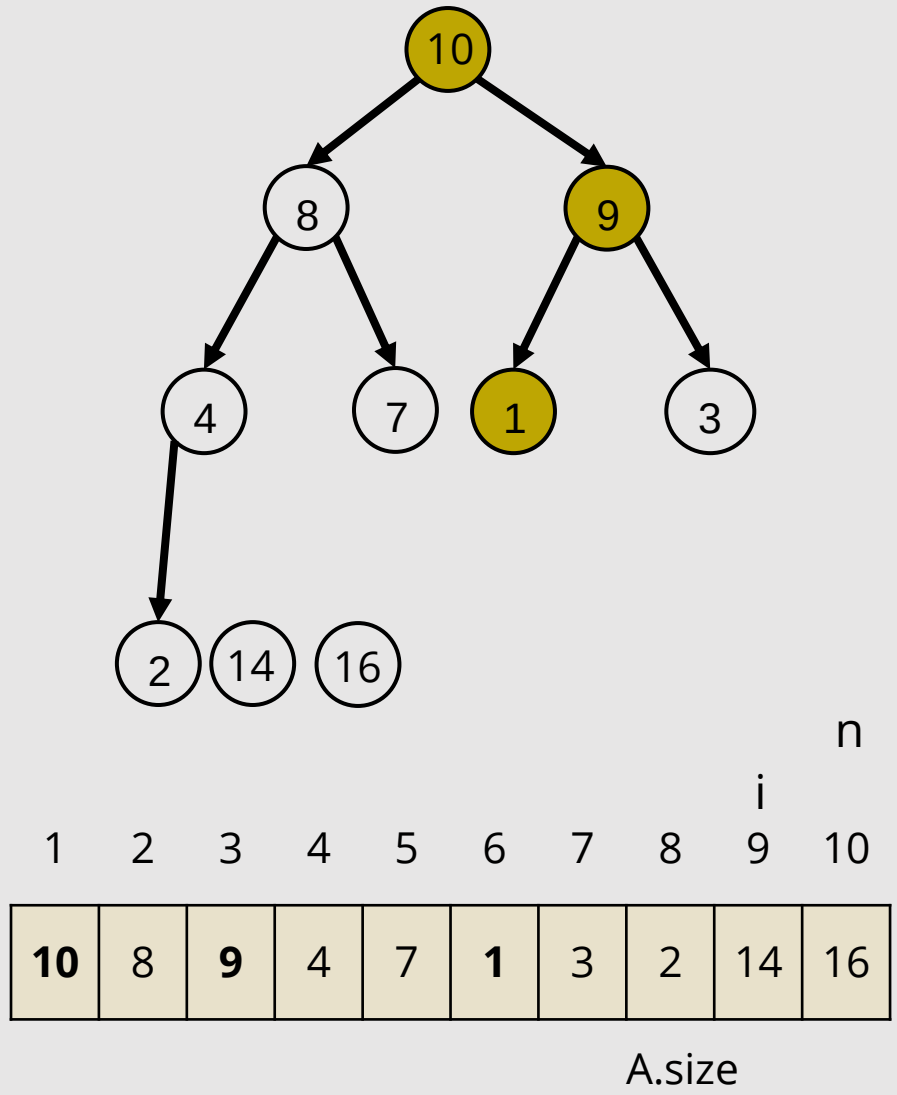
HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. ***A.size* = *A.size* − 1**
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**



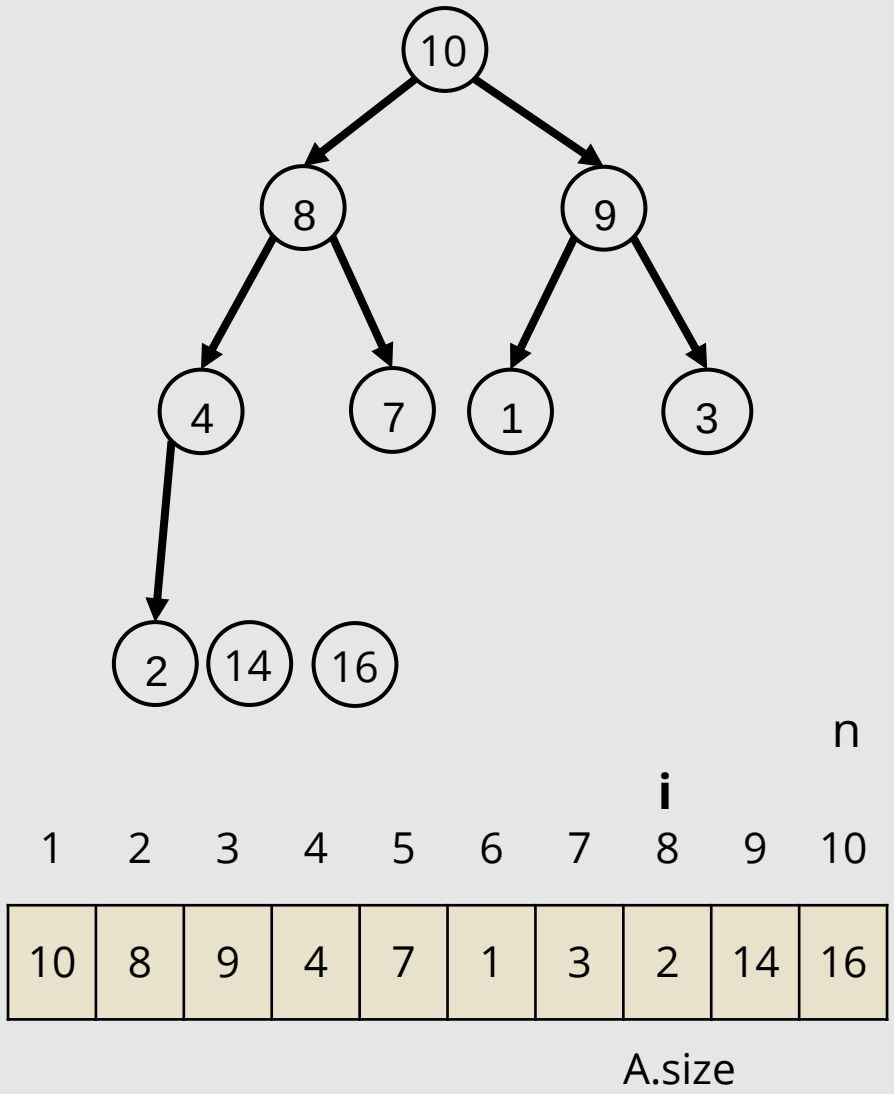
HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. **MAX-HEAPIFY(A, 1)**
8. **end for**
9. **end function**



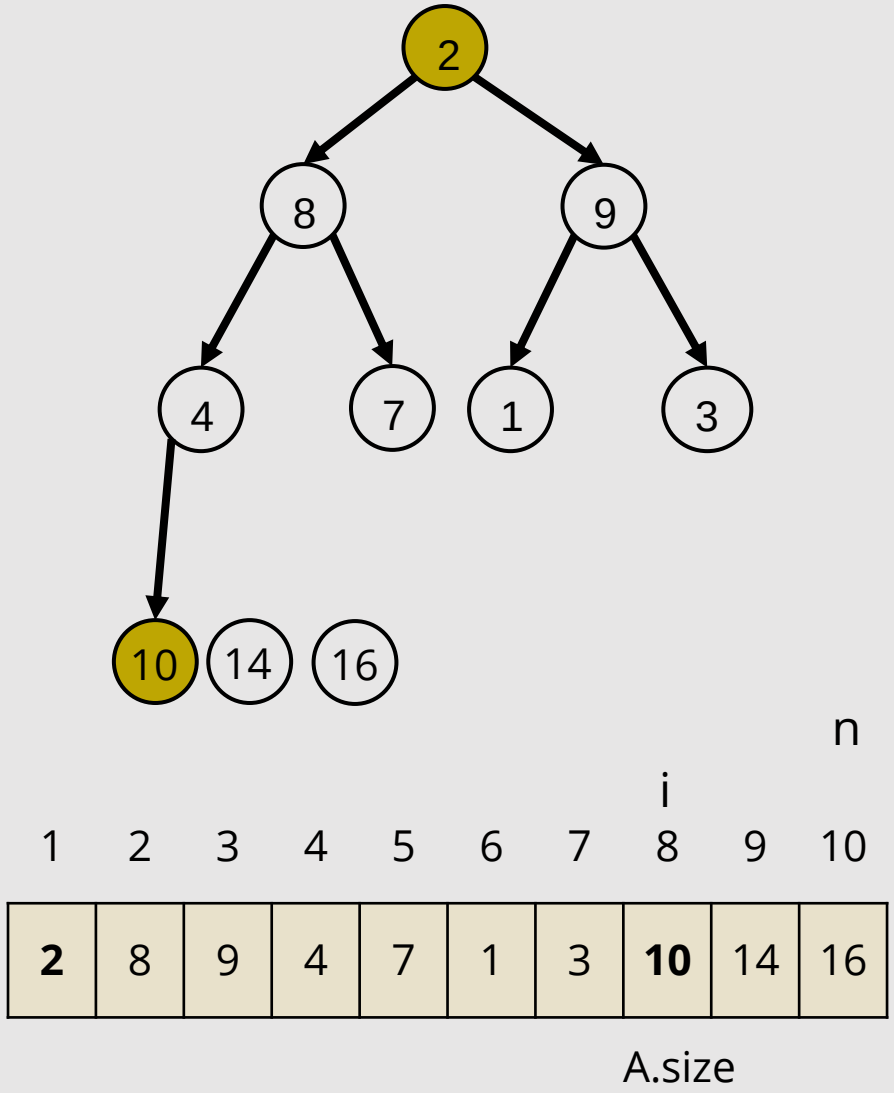
HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** ***i*** **=** ***n*** **downto 2 do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**



HEAPSORT BEISPIEL

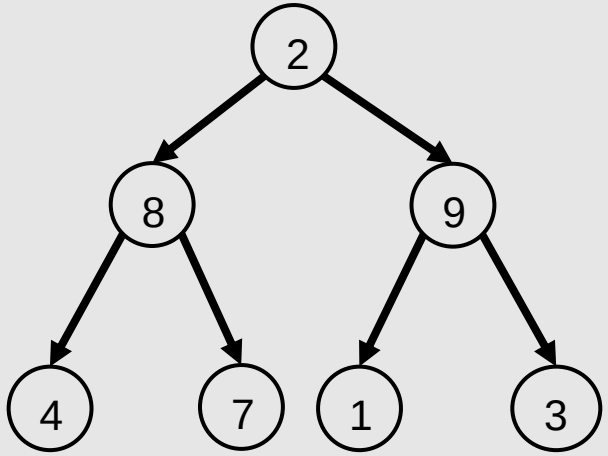
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. **vertausche** ***A*[1] ↔ *A*[*i*]**
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

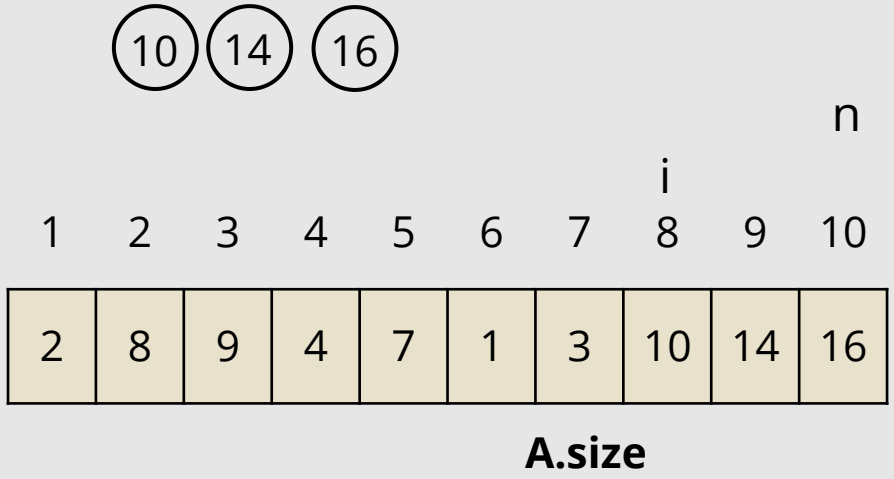


58

HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. ***A.size* = *A.size* − 1**
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

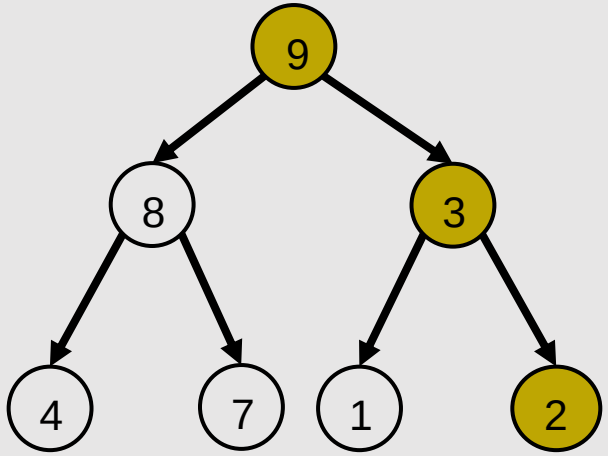


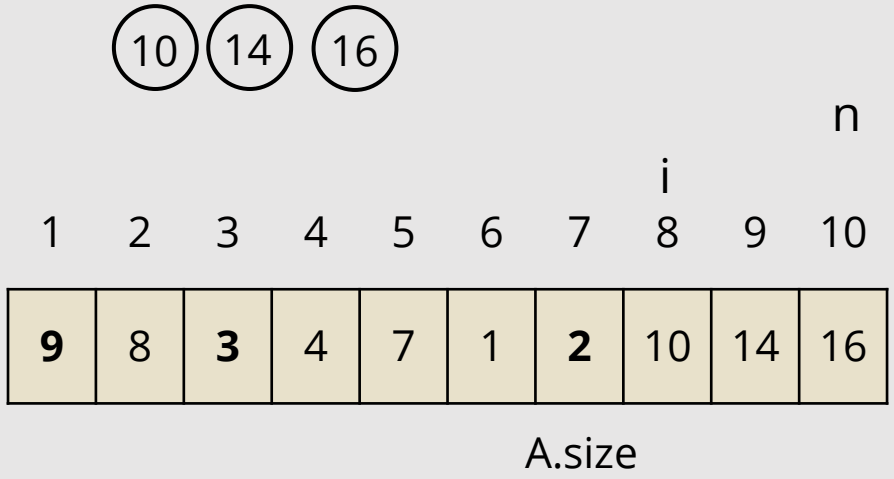


59

HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. **MAX-HEAPIFY(A, 1)**
8. **end for**
9. **end function**

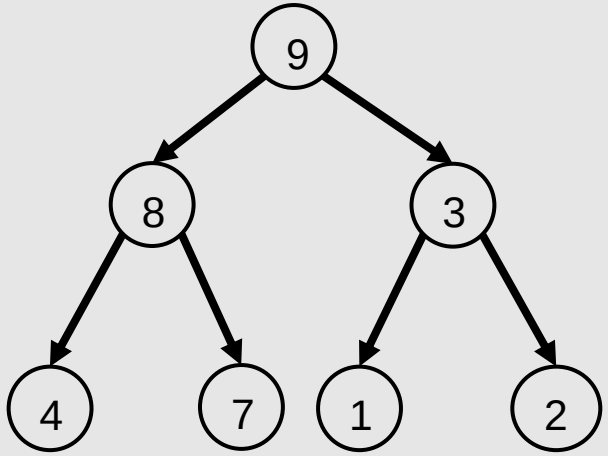


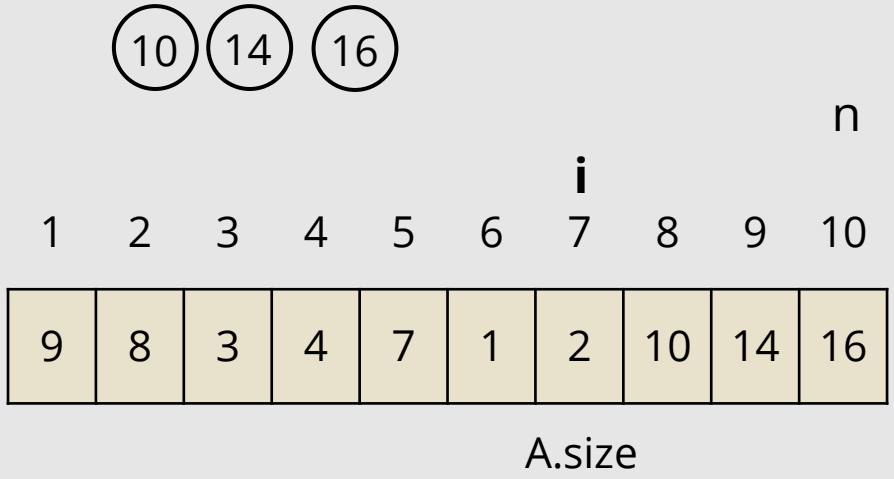


60

HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** ***i*** **=** ***n*** **downto 2 do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

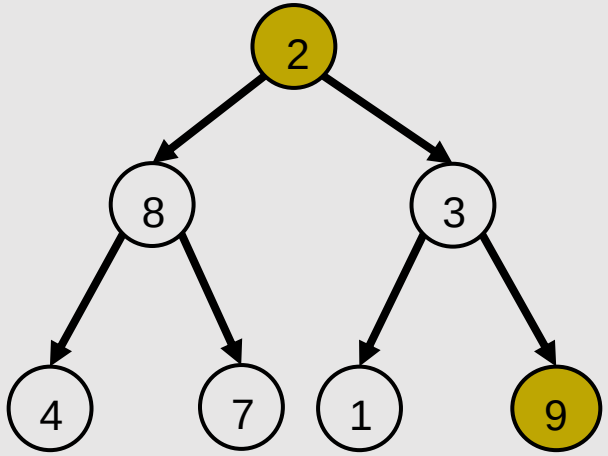


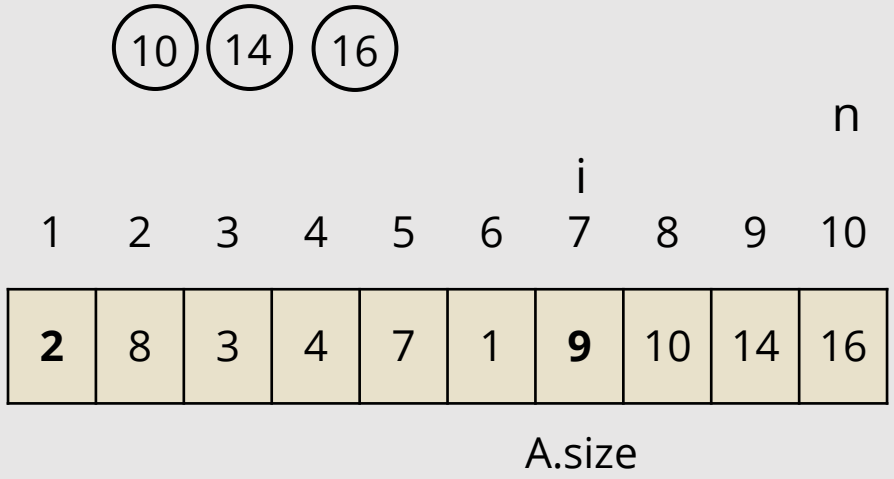


61

HEAPSORT BEISPIEL

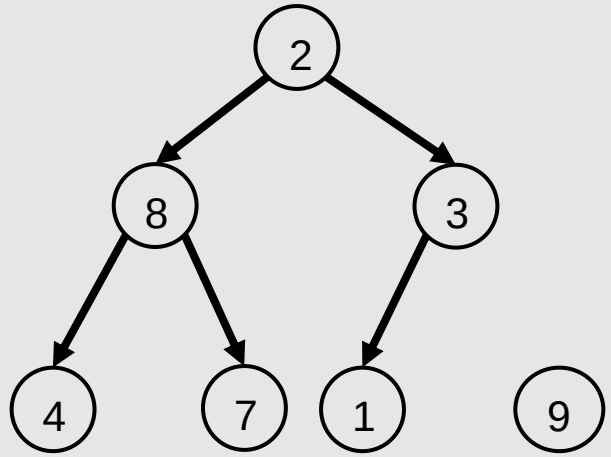
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. **vertausche** ***A*[1] ↔ *A*[*i*]**
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

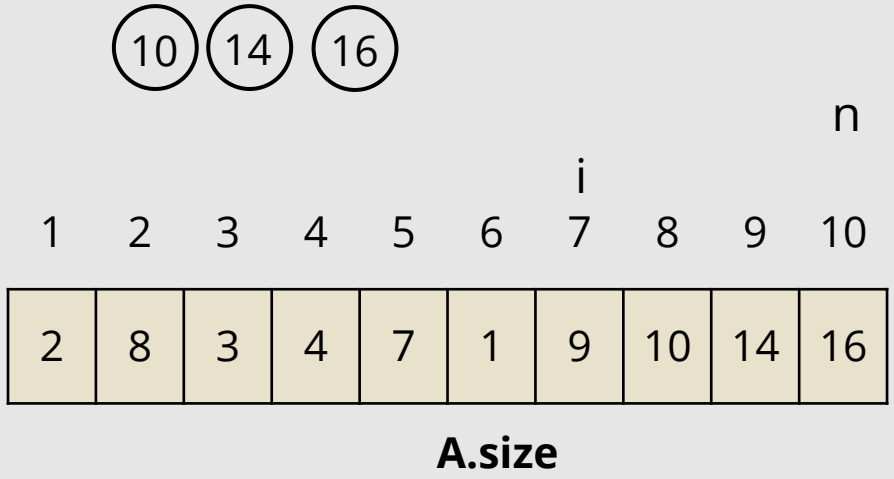




HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. ***A.size* = *A.size* − 1**
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

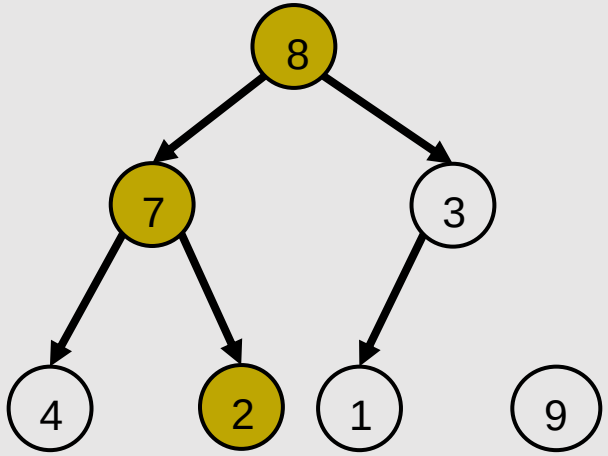


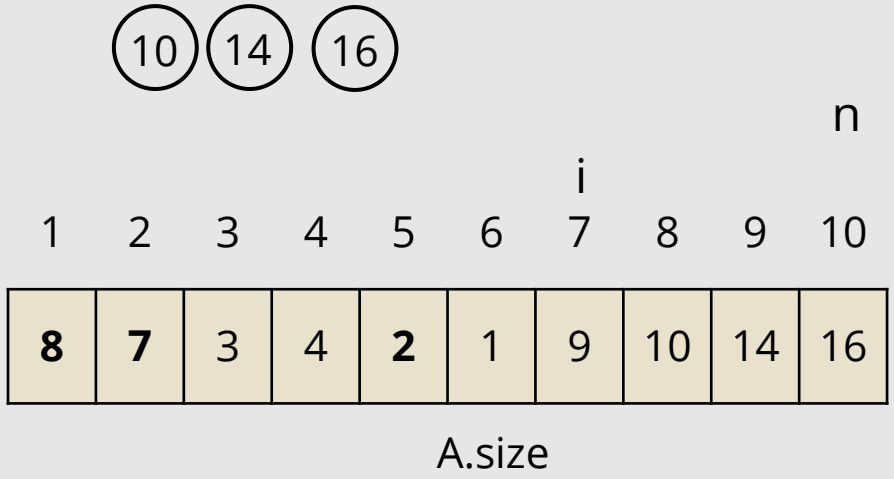


63

HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. **MAX-HEAPIFY(A, 1)**
8. **end for**
9. **end function**

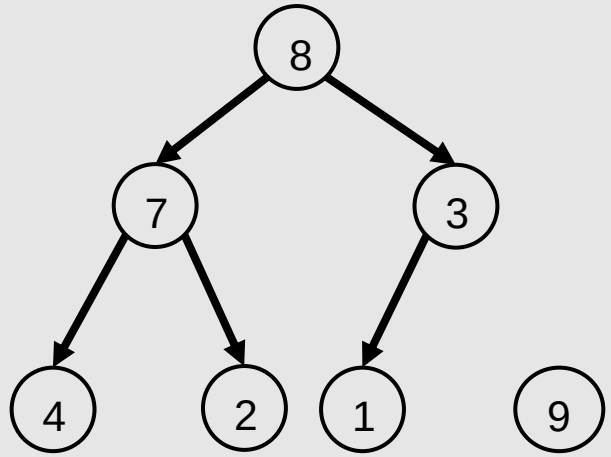


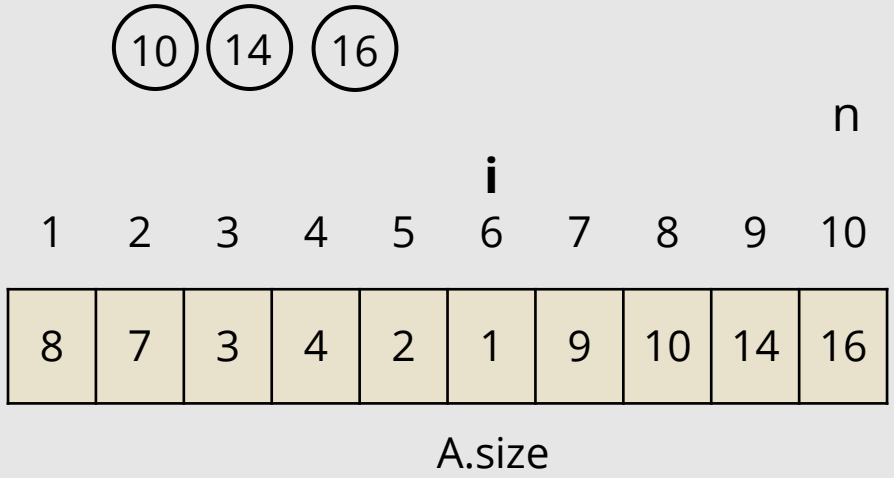


64

HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** ***i*** **=** ***n*** **downto 2 do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

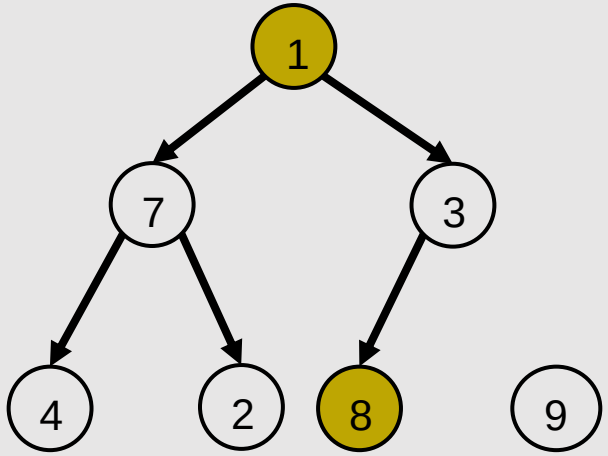


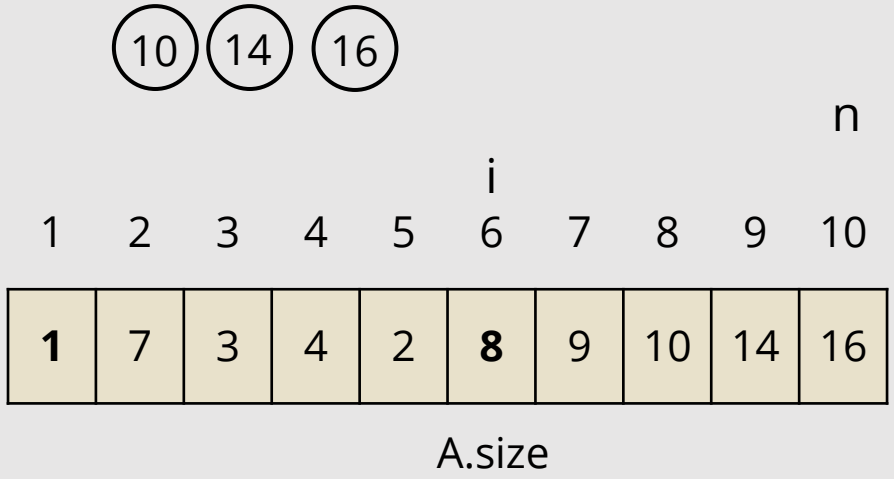


65

HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. **vertausche** ***A*[1] ↔ *A*[*i*]**
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

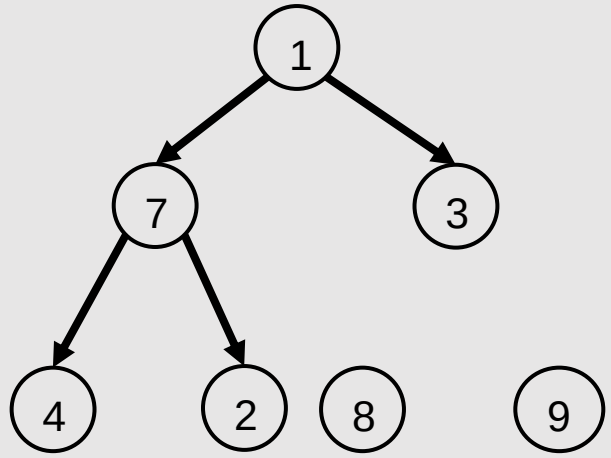


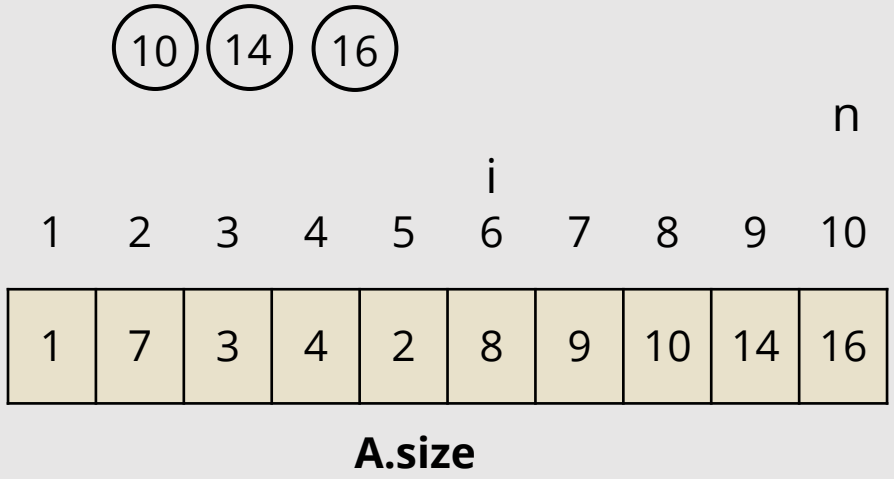


66

HEAPSORT BEISPIEL

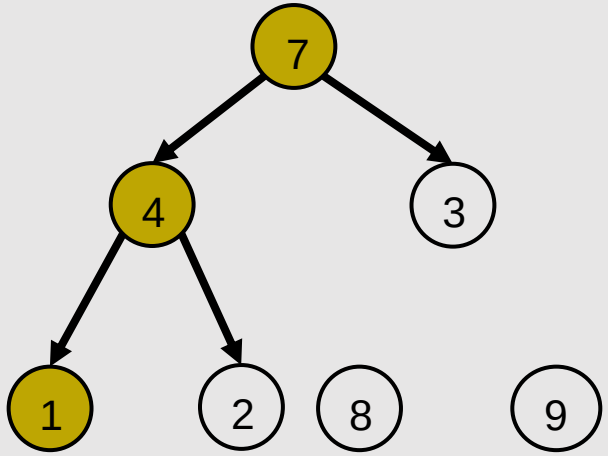
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. ***A.size* = *A.size* − 1**
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

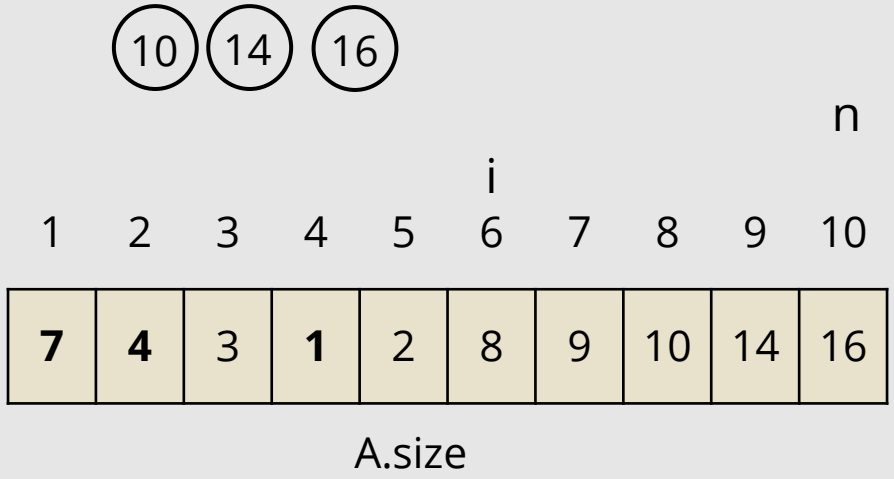




HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. **MAX-HEAPIFY(A, 1)**
8. **end for**
9. **end function**

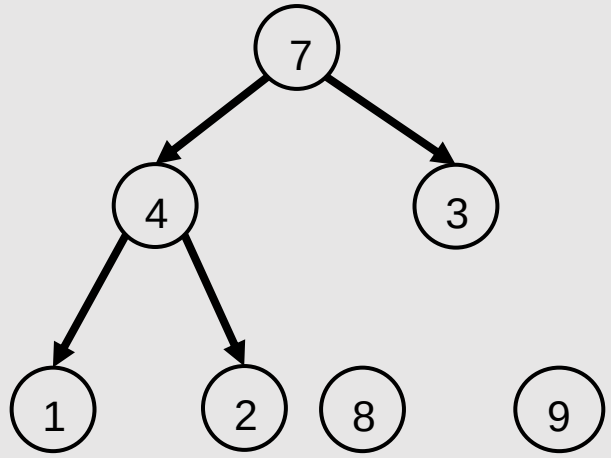


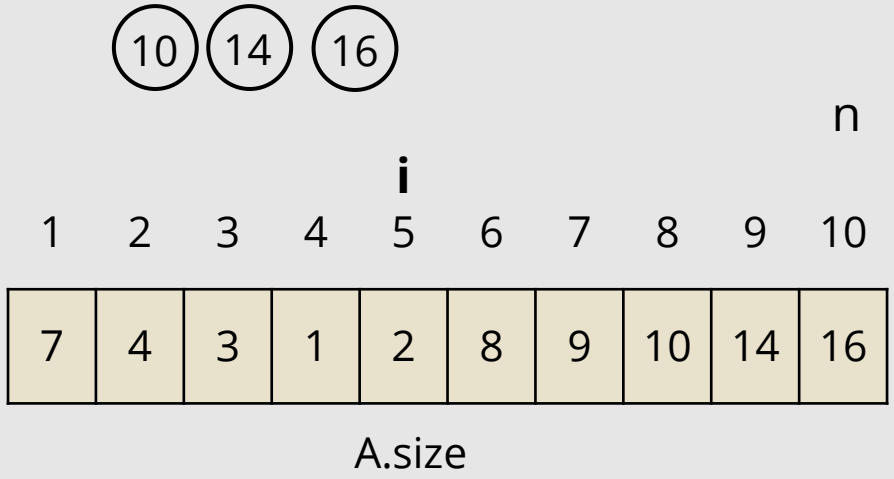


68

HEAPSORT BEISPIEL

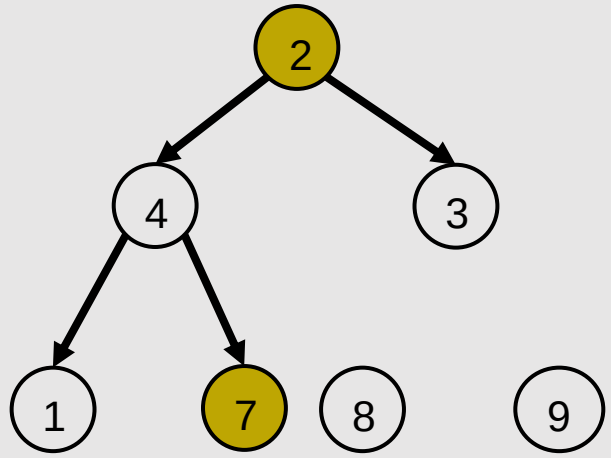
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** ***i*** **=** ***n*** **downto 2 do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

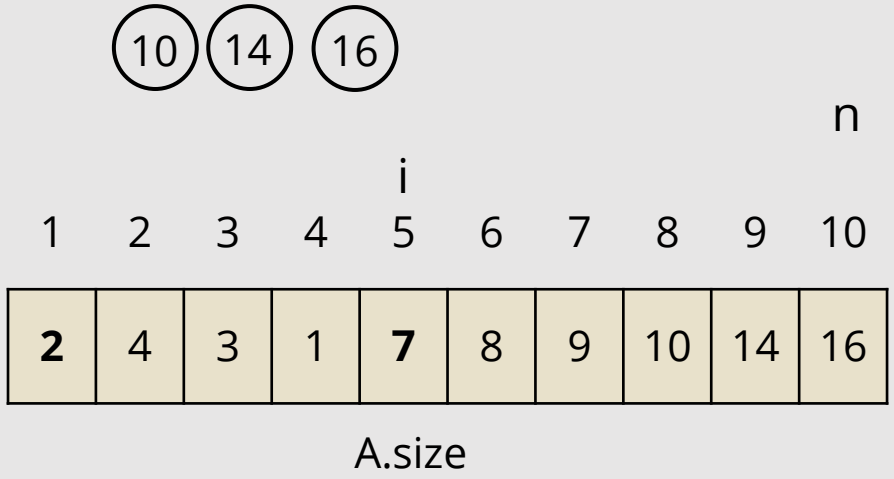




HEAPSORT BEISPIEL

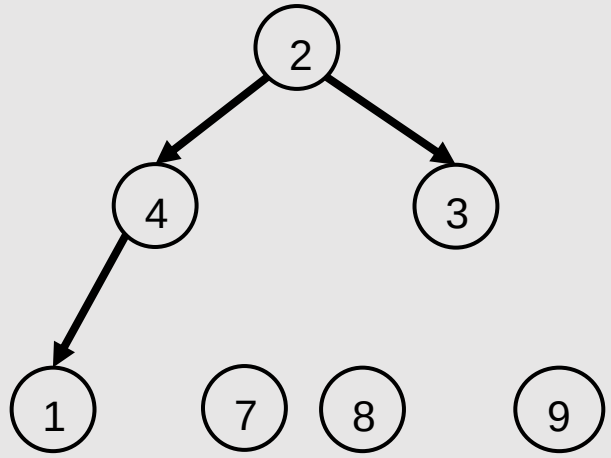
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. **vertausche** ***A*[1] ↔ *A*[*i*]**
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

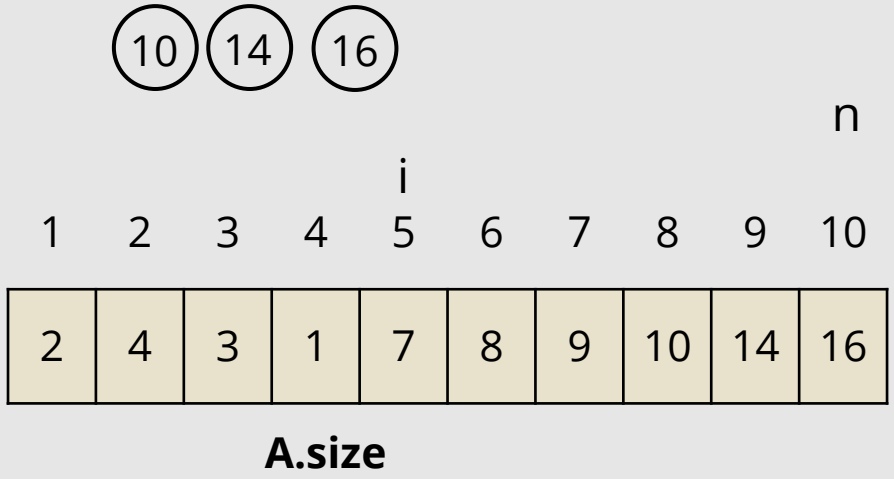




HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. ***A.size* = *A.size* − 1**
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

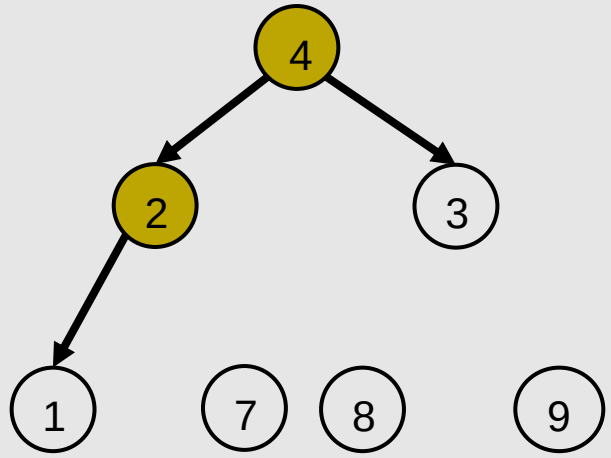


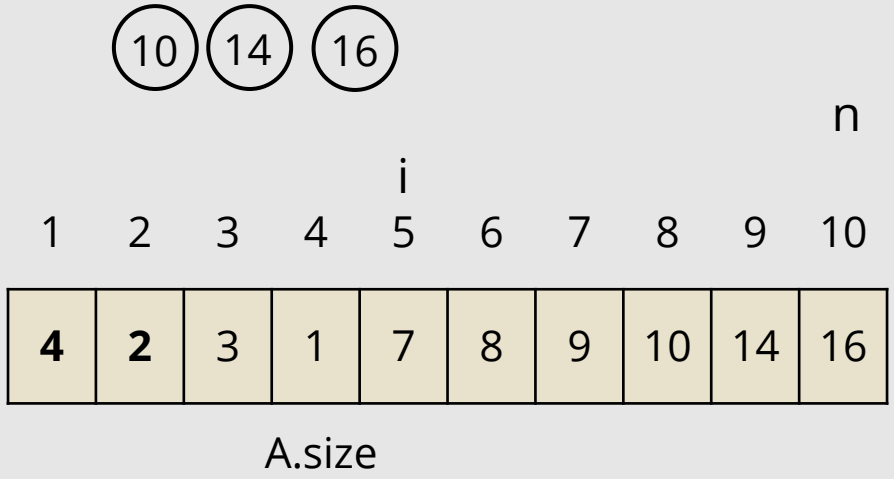


71

HEAPSORT BEISPIEL

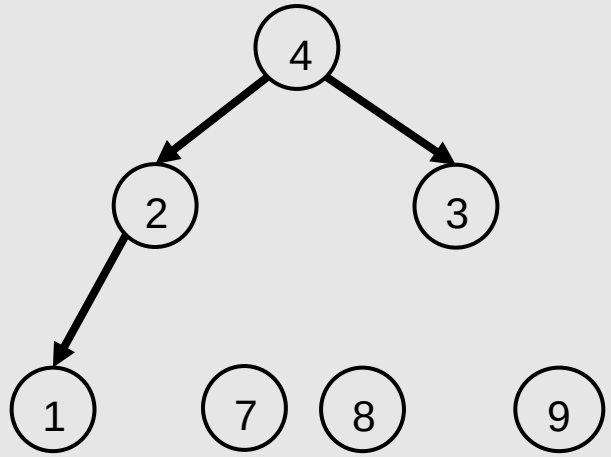
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. **MAX-HEAPIFY(A, 1)**
8. **end for**
9. **end function**

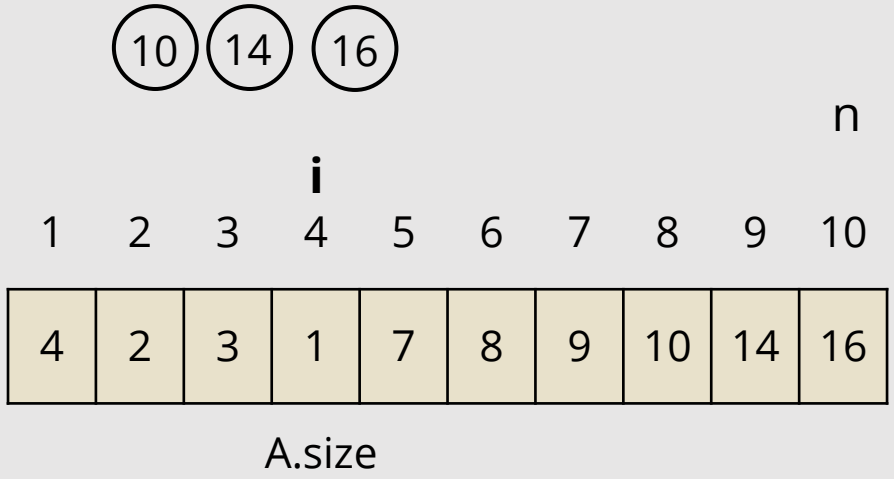




HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** ***i*** **=** ***n*** **downto 2 do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

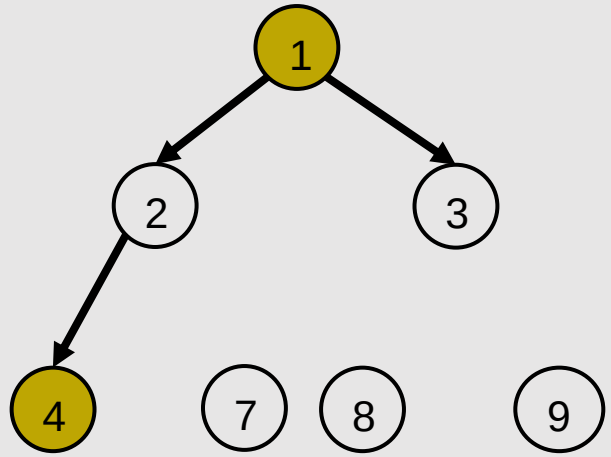


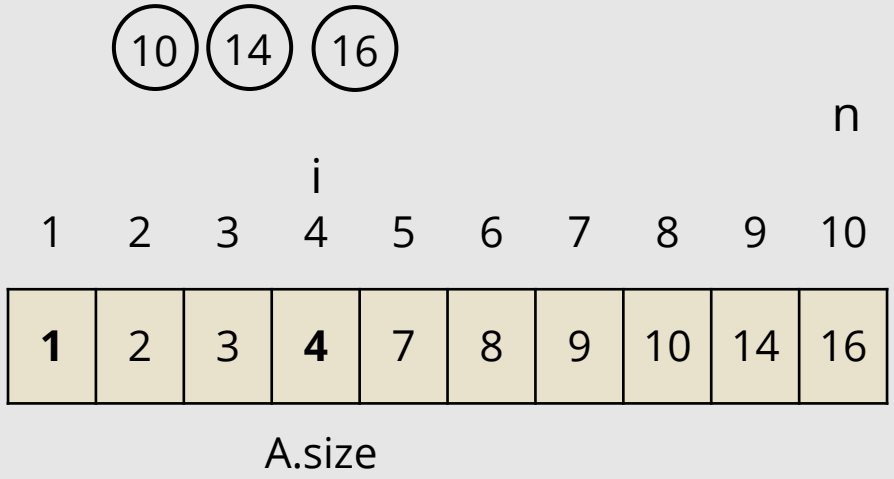


73

HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. **vertausche** ***A*[1] ↔ *A*[*i*]**
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

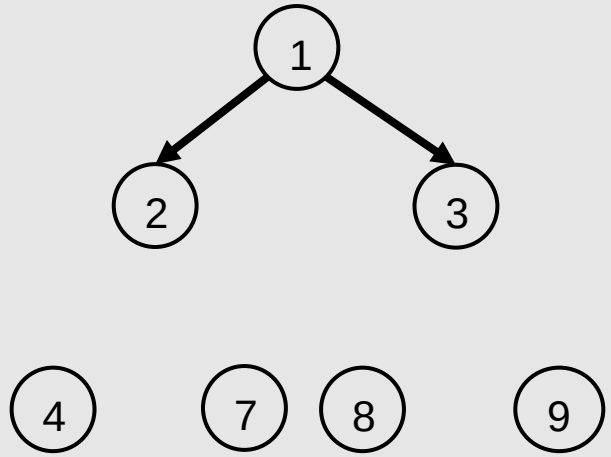




74

HEAPSORT BEISPIEL

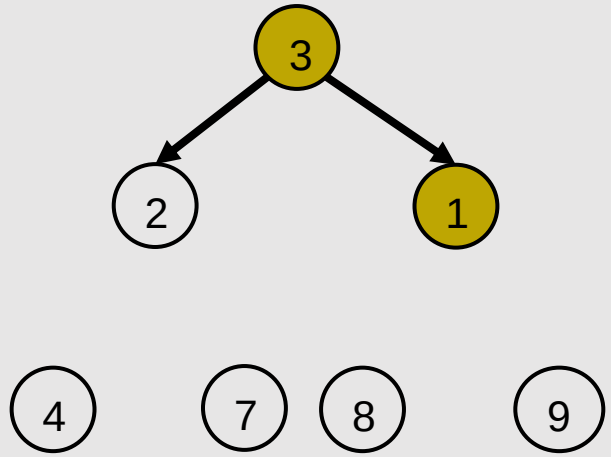
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. ***A.size* = *A.size* − 1**
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

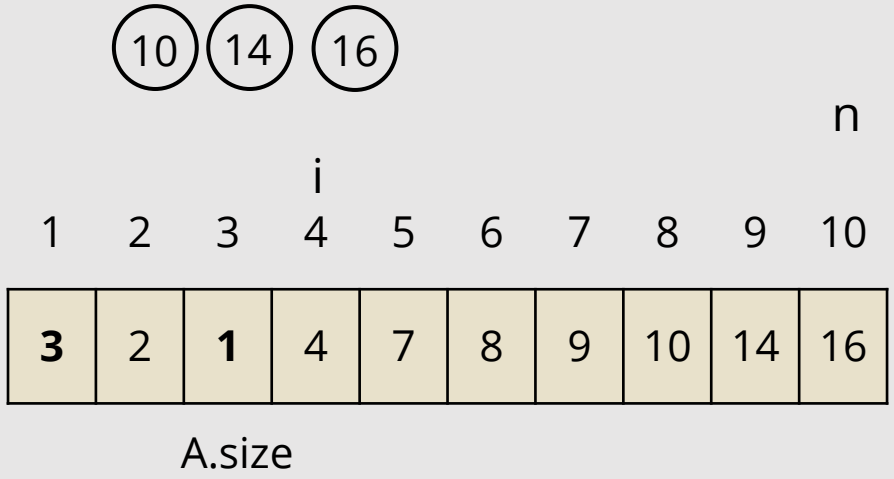


75

HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. **MAX-HEAPIFY(A, 1)**
8. **end for**
9. **end function**

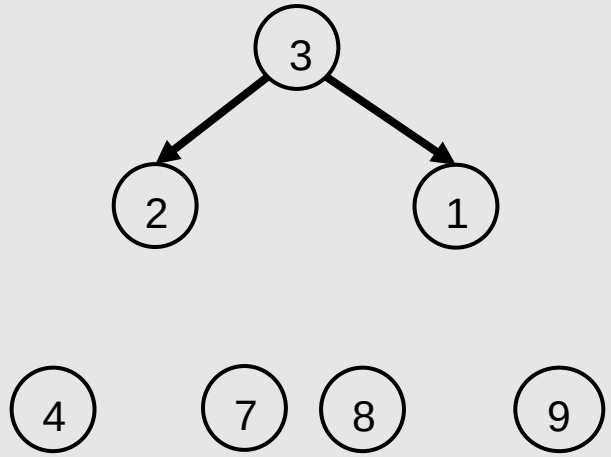


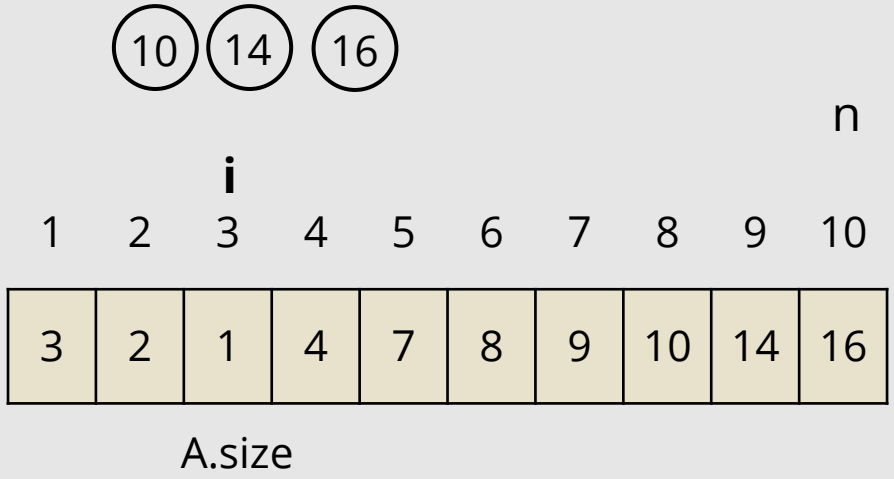


76

HEAPSORT BEISPIEL

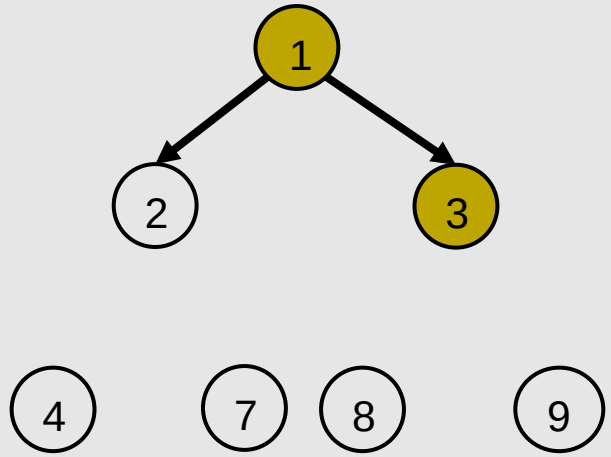
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** ***i*** **=** ***n*** **downto 2 do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

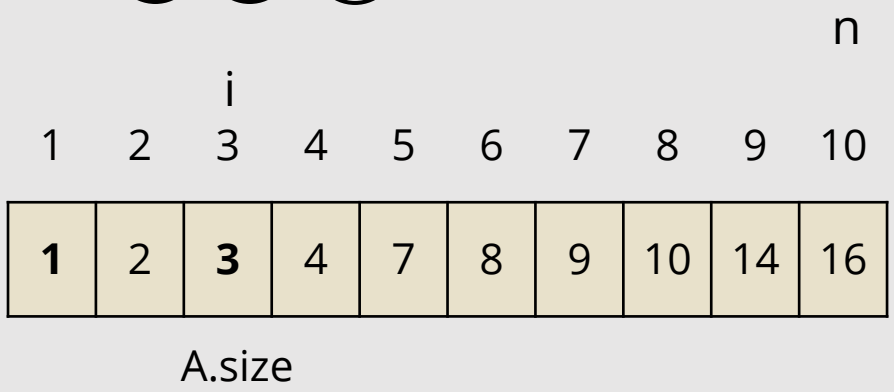




HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. **vertausche** ***A*[1] ↔ *A*[*i*]**
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**

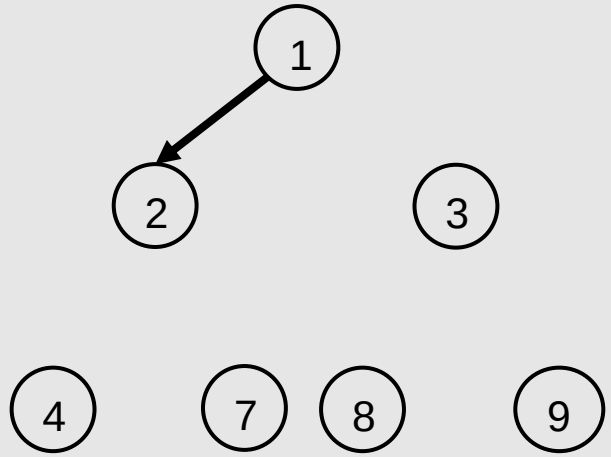


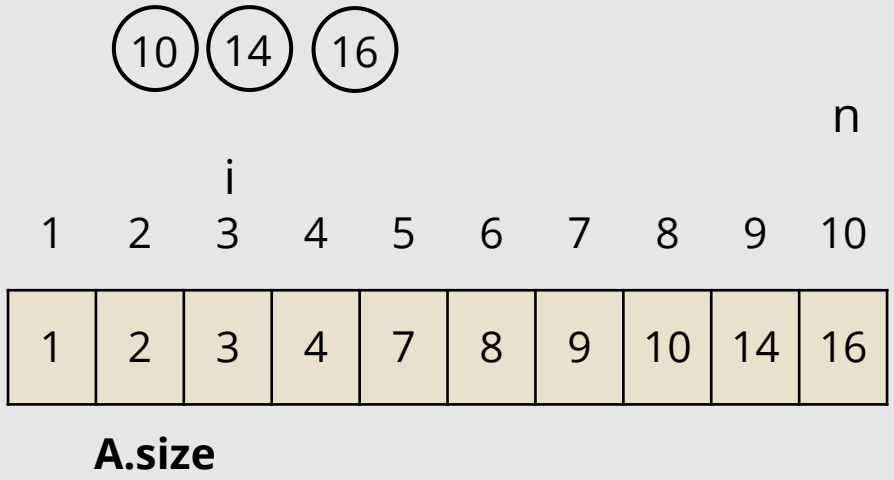


78

HEAPSORT BEISPIEL

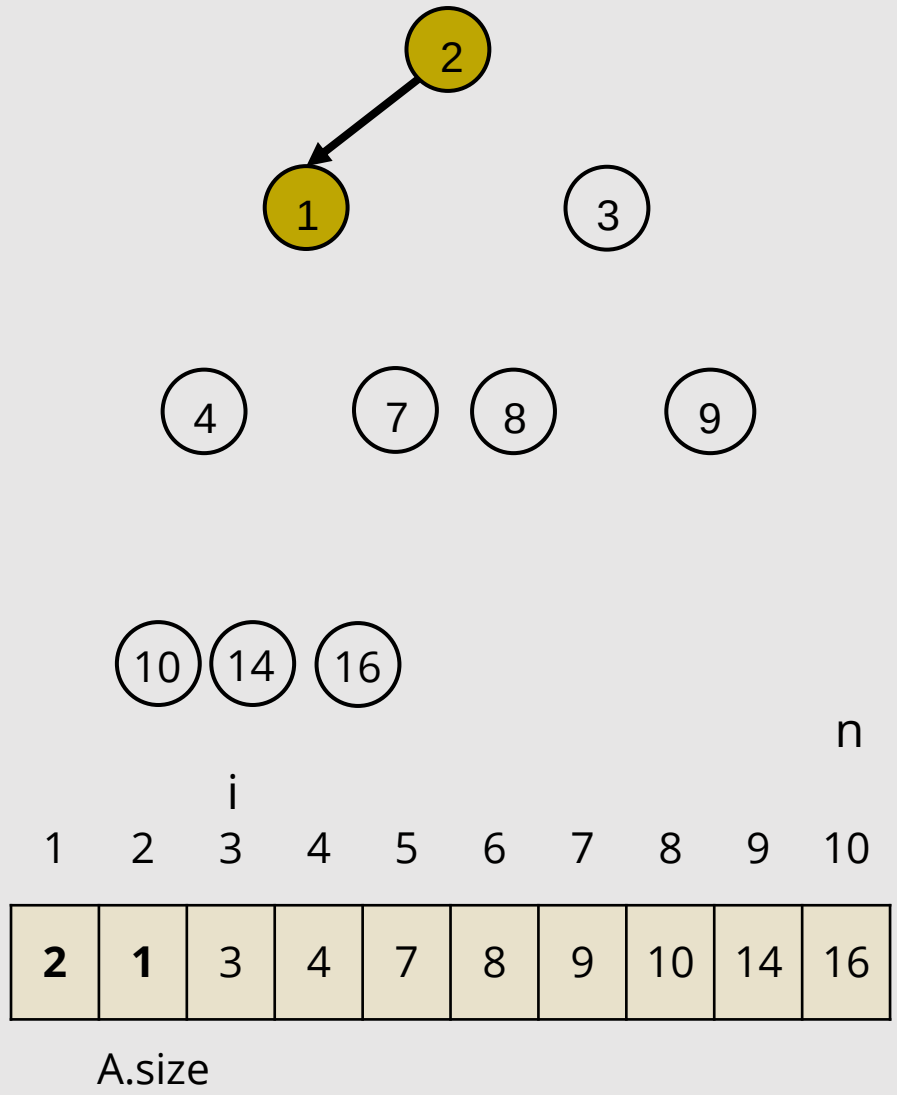
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n = A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. ***A.size = A.size* − 1**
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**





HEAPSORT BEISPIEL

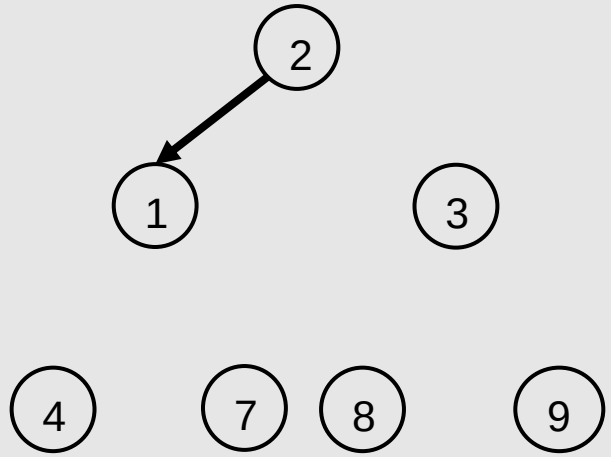
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. **MAX-HEAPIFY(A, 1)**
8. **end for**
9. **end function**

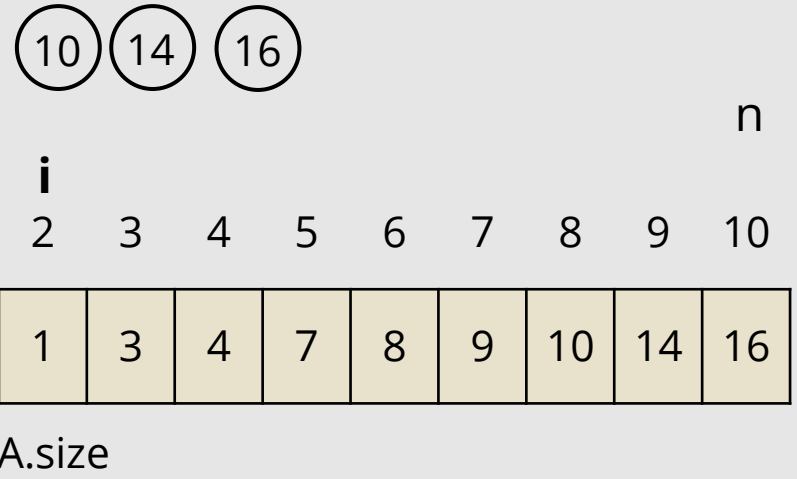


80

HEAPSORT BEISPIEL

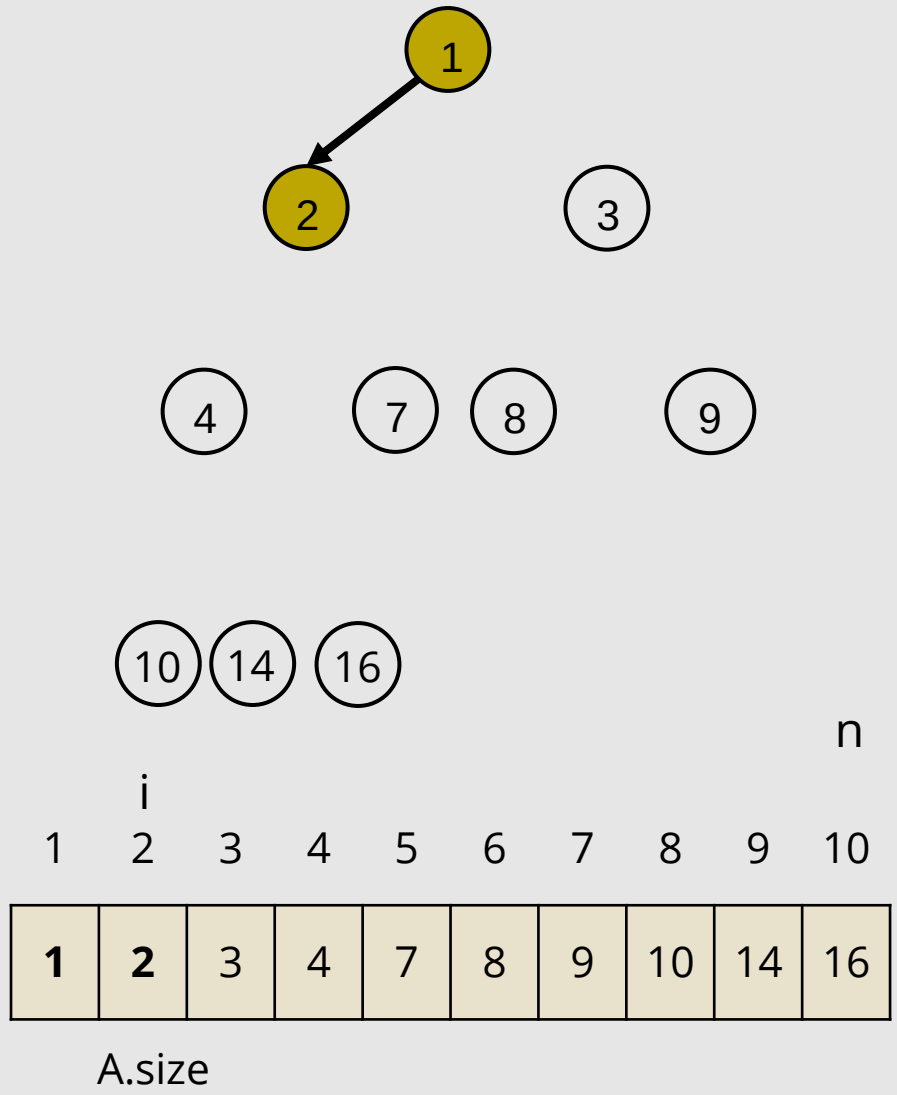
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** ***i*** **=** ***n*** **downto 2 do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**





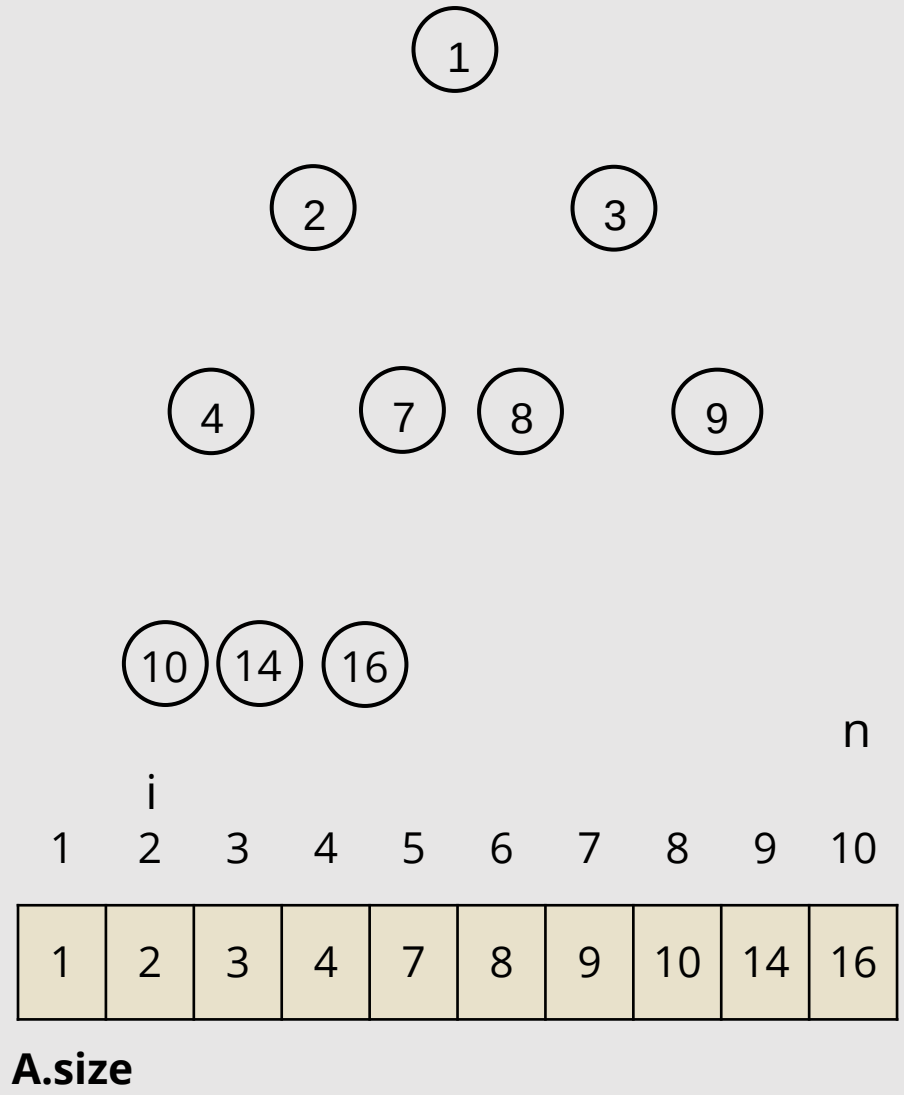
HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. **vertausche** ***A*[1] ↔ *A*[*i*]**
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**



HEAPSORT BEISPIEL

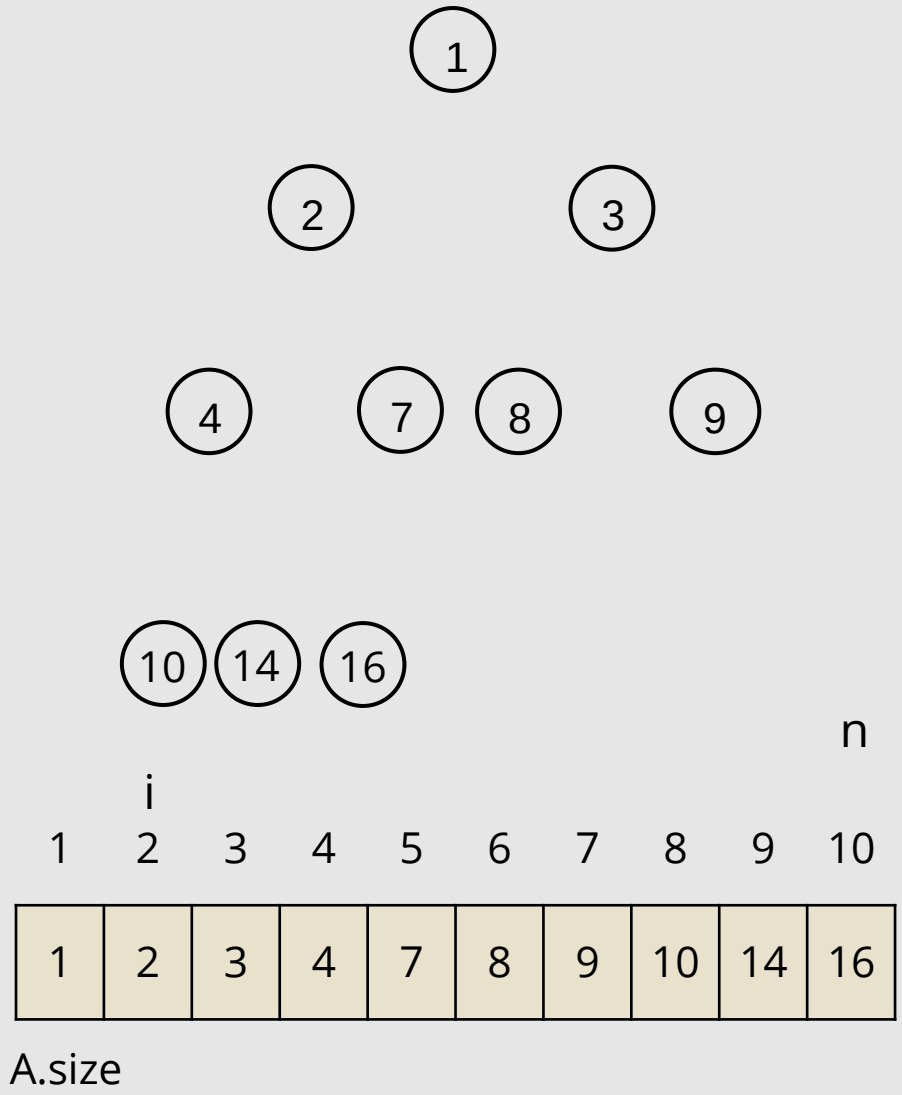
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n = A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. ***A.size = A.size* − 1**
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**



83

HEAPSORT BEISPIEL

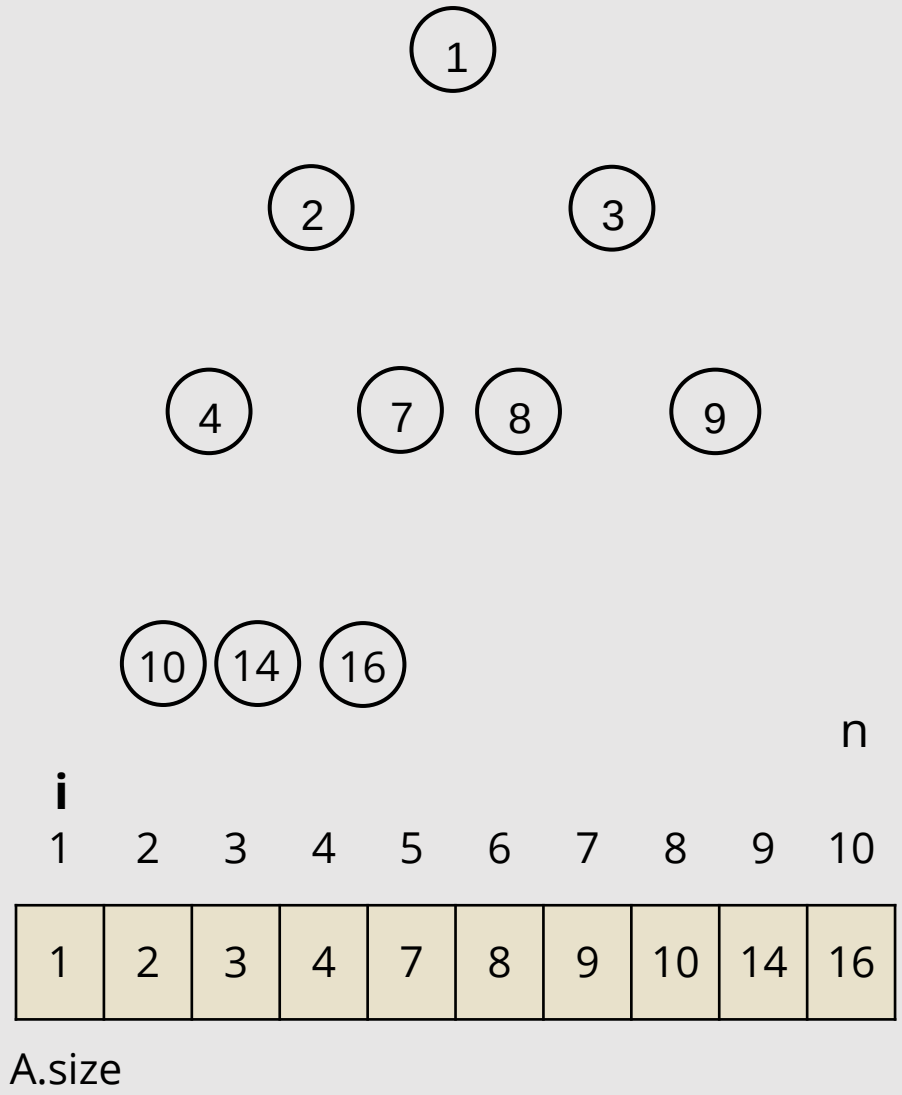
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. **MAX-HEAPIFY(A, 1)**
8. **end for**
9. **end function**



84

HEAPSORT BEISPIEL

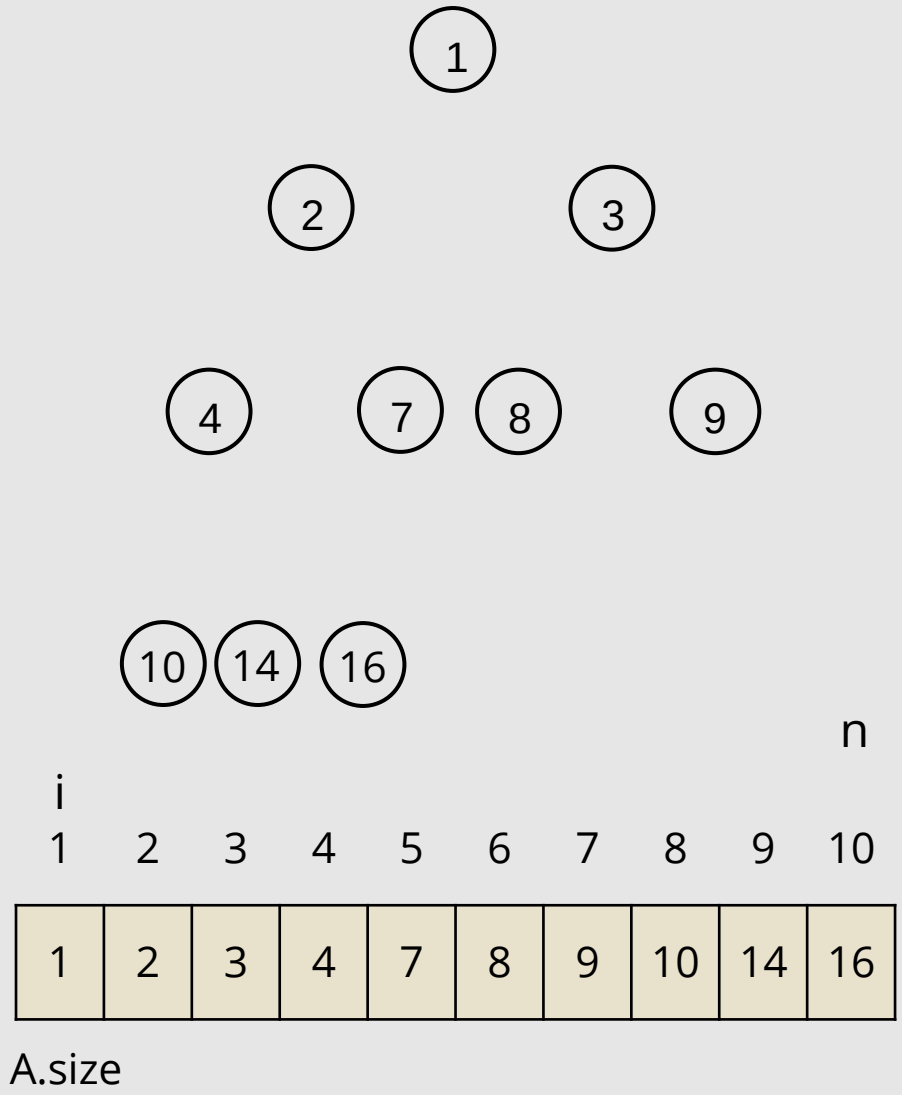
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** ***i*** **=** ***n*** **downto 2 do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**



85

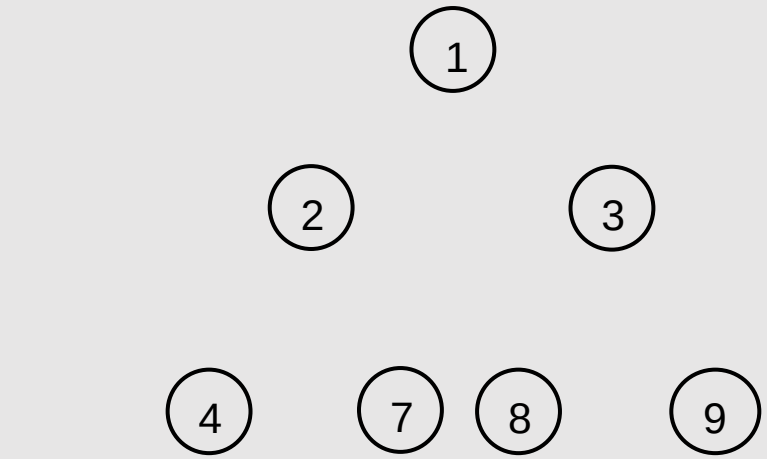
HEAPSORT BEISPIEL

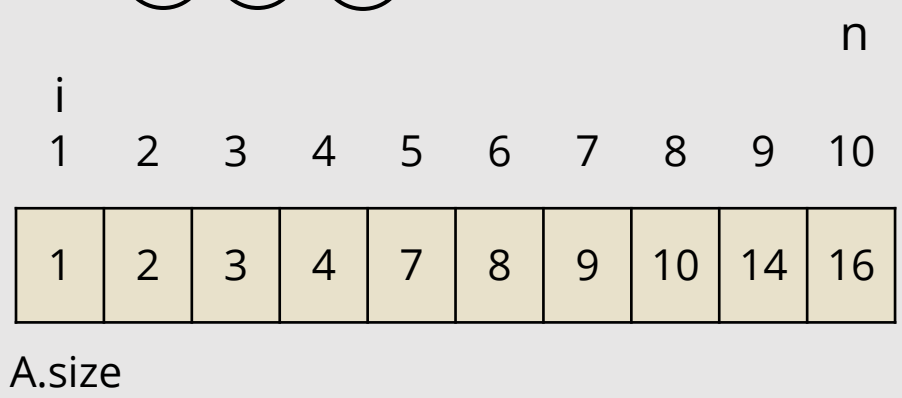
1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**



HEAPSORT BEISPIEL

1. **function** HEAPSORT(A)
2. BUILD-MAX-HEAP(A)
3. *n* = *A.size*
4. **for** *i* = *n* downto 2 **do**
5. vertausche *A*[1] ↔ *A*[*i*]
6. *A.size* = *A.size* − 1
7. MAX-HEAPIFY(A, 1)
8. **end for**
9. **end function**





LAUFZEIT VON HEAPSORT

- BUILD-MAX-HEAP hat $O(n)$
- $n - 1$ Aufrufe von Max-Heapify ergeben $O(n \cdot \log n)$
- Laufzeit ist also $T(n) = O(n \cdot \log n)$.
- Man kann auch zeigen, dass die best-case und die worst-case Laufzeit bei

  $T(n) = \Omega(n \cdot \log n)$ liegen.
- Heapsort ist ein exzellenter Algorithmus, wird aber in der Praxis bei guter Implementation durch Quicksort geschlagen.

88