Datenstrukturen und Algorithmen

Vorlesung 8: Heapsort (K6)

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

https://moves.rwth-aachen.de/teaching/ss-18/dsal/

14. Mai 2018

Übersicht

1. Heaps
2. Heapaufbau

   naiv

   effizienteres
3. Heapsort
4. Anwendung: Prioritätswarteschlangen

Übersicht

Heaps

Heap (Haufen)

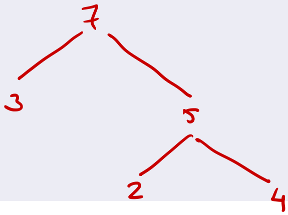
Ein Heap ist ein Binärbaum, der Elemente mit Schlüsseln enthält und in ein Array eingebettet ist. Die Heap-Bedingung für *Max-Heap*s fordert:

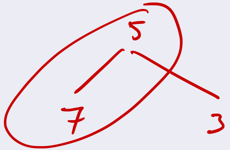
Heaps

Heap (Haufen)

Ein Heap ist ein Binärbaum, der Elemente mit Schlüsseln enthält und in ein Array eingebettet ist. Die Heap-Bedingung für *Max-Heap*s fordert:

- Der Schlüssel eines Knotens ist stets größer als (bzw. mindestens so groß wie) die Schlüssel seiner Kinder.





kein Maxheap

Heaps

Heap (Haufen)

Ein Heap ist ein Binärbaum, der Elemente mit Schlüsseln enthält und in ein Array eingebettet ist. Die Heap-Bedingung für *Max-Heap*s fordert:

- Der Schlüssel eines Knotens ist stets größer als (bzw. mindestens so groß wie) die Schlüssel seiner Kinder.

Weiter gilt:

- Alle Ebenen, abgesehen von evtl. der untersten, sind komplett gefüllt.

Heaps

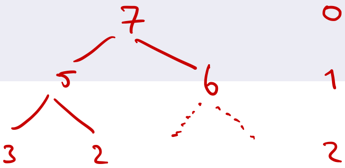
Heap (Haufen)

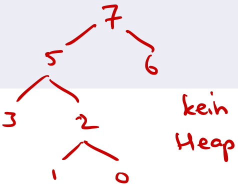
Ein Heap ist ein Binärbaum, der Elemente mit Schlüsseln enthält und in ein Array eingebettet ist. Die Heap-Bedingung für *Max-Heaps* fordert:

- Der Schlüssel eines Knotens ist stets größer als (bzw. mindestens so groß wie) die Schlüssel seiner Kinder.

Weiter gilt:

- Alle Ebenen, abgesehen von evtl. der untersten, sind komplett gefüllt.
- Die Blätter befinden sich damit alle auf einer (höchstens zwei) Ebene(n).





Heaps

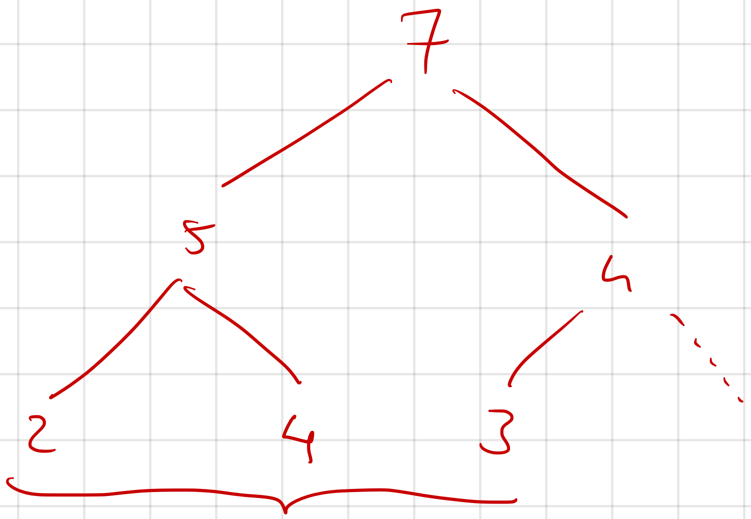
Heap (Haufen)

Ein Heap ist ein Binärbaum, der Elemente mit Schlüsseln enthält und in ein Array eingebettet ist. Die Heap-Bedingung für *Max-Heap*s fordert:

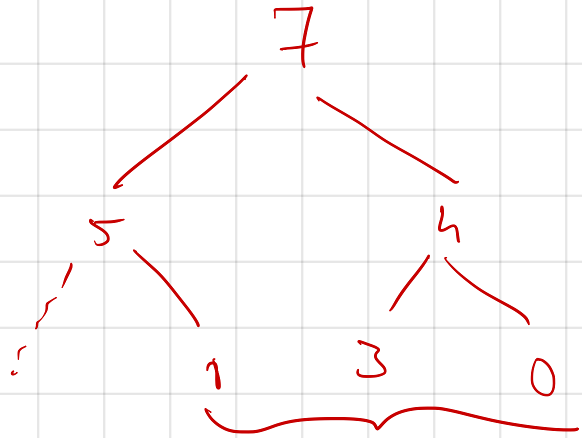
- Der Schlüssel eines Knotens ist stets größer als (bzw. mindestens so groß wie) die Schlüssel seiner Kinder.

Weiter gilt:

- Alle Ebenen, abgesehen von evtl. der untersten, sind komplett gefüllt.
- Die Blätter befinden sich damit alle auf einer (höchstens zwei) Ebene(n).
- Die Blätter der untersten Ebene sind linksbündig angeordnet.



max Heap



Kein max Heap

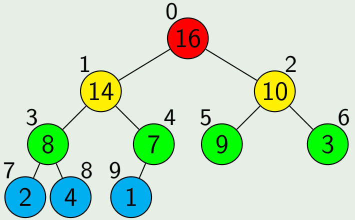
Arrayeinbettung eines Heaps

Arrayeinbettung

Das Array ~~a~~ E wird wie folgt als Binärbaum aufgefasst:

- Die Wurzel liegt in `E[0]`.

Beispiel



16	14	10	8	7	9	3	2	4	1
0	1	2	3	4	5	6	7	8	9

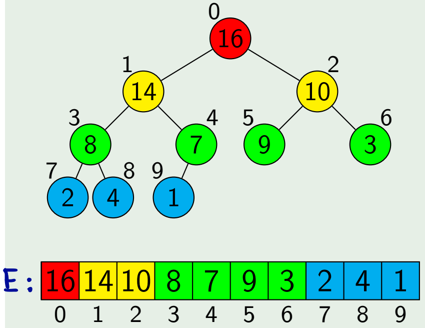
Arrayeinbettung eines Heaps

Arrayeinbettung

Das Array `a` wird wie folgt als Binärbaum aufgefasst:

- Die Wurzel liegt in `E[0]`.
- Das linke Kind von `E[i]` liegt in `E[2 * i + 1]`.

Beispiel



$i = 3$

$2 * i + 1 = 7$

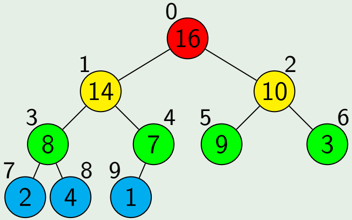
Arrayeinbettung eines Heaps

Arrayeinbettung

Das Array a wird wie folgt als Binärbaum aufgefasst:

- Die Wurzel liegt in `E[0]`.
- Das linke Kind von `E[i]` liegt in `E[2 * i + 1]`.
- Das rechte Kind von `E[i]` liegt in `E[2 * i + 2]`.

Beispiel



16	14	10	8	7	9	3	2	4	1
0	1	2	3	4	5	6	7	8	9

$i=3$ $\quad 2i+1 \quad 2i+2 = 8$

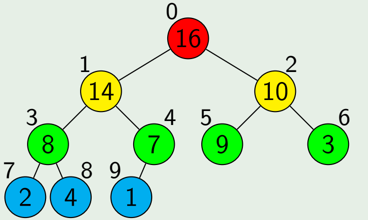
Arrayeinbettung eines Heaps

Arrayeinbettung

Das Array `a` wird wie folgt als Binärbaum aufgefasst:

- Die Wurzel liegt in `E[0]`.
- Das linke Kind von `E[i]` liegt in `E[2 * i + 1]`.
- Das rechte Kind von `E[i]` liegt in `E[2 * i + 2]`.

Beispiel



16	14	10	8	7	9	3	2	4	1
0	1	2	3	4	5	6	7	8	9

- Durch die möglichst vollständige Füllung der Ebenen werden „Löcher“ im Array vermieden.

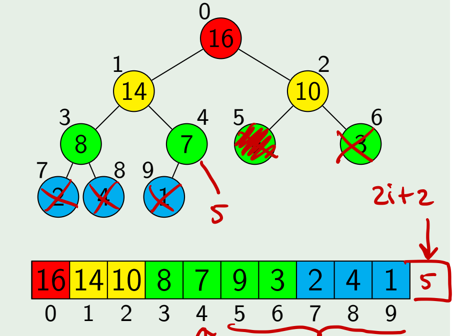
Arrayeinbettung eines Heaps

Arrayeinbettung

Das Array a wird wie folgt als Binärbaum aufgefasst:

- Die Wurzel liegt in E[0].
- Das linke Kind von a[i] liegt in E[2 * i + 1].
- Das rechte Kind von a[i] liegt in E[2 * i + 2].

Beispiel

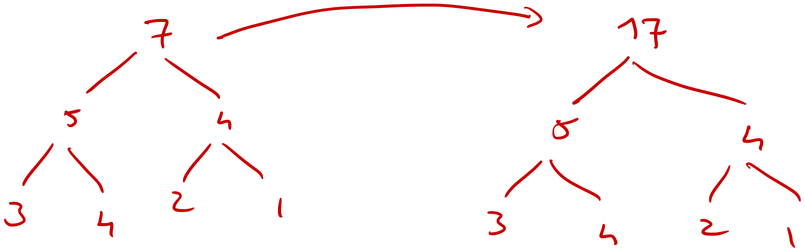


- Durch die möglichst vollständige Füllung der Ebenen werden „Löcher" im Array vermieden.
- Vergrößert man den Baum um ein Element, so wird das Array gerade um ein Element länger.

Heaps – Eigenschaften

Lemma

Vergrößert man den Schlüssel der Wurzel, dann bleibt der Baum ein Heap.



Heaps – Eigenschaften

$n =$ Anzahl der Elemente im Heap

Lemma

Vergrößert man den Schlüssel der Wurzel, dann bleibt der Baum ein Heap.

Lemma

Jedes Array „ist ein Heap“ ab Position $\lfloor \frac{n}{2} \rfloor$.

$$E = 16 \quad 14 \quad 10 \quad 8 \quad 7 \quad \underbrace{9 \quad 3 \quad 2 \quad 4 \quad 1}_{*} \qquad n = 10$$

Heaps – Eigenschaften

Lemma

Vergrößert man den Schlüssel der Wurzel, dann bleibt der Baum ein Heap.

Lemma

Jedes Array „ist ein Heap" ab Position $\lfloor \frac{n}{2} \rfloor$.

- Ein Heap hat $\lfloor \frac{n}{2} \rfloor$ innere Knoten.

$E = \underbrace{16 \quad 14 \quad 10 \quad 8 \quad 7}_{5 \text{ innere Knoten}} \quad 9 \quad 3 \quad 2 \quad 4 \quad 1 \qquad n=10$

Lemma: $\boxed{\text{ein Heap mit } n \text{ Elementen hat } \left\lceil \frac{n}{2} \right\rceil \text{ Blätter}}$

Beweis: sei H der Höhe der Heap Anzahl der Ebenen

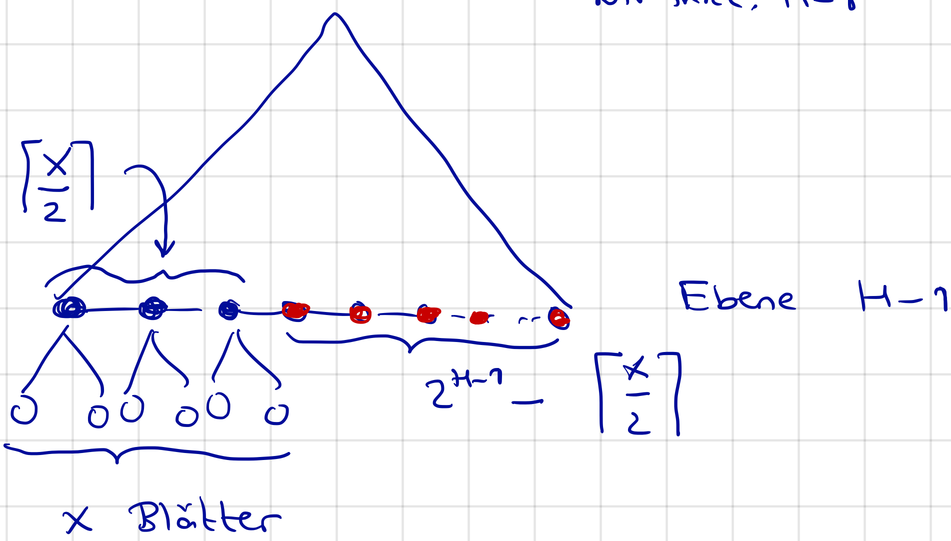
alle Blätter befinden sich auf Ebene H und H−1

Ebene H−1 ist voll besetzt

Ebene H muss nicht voll besetzt sein

Sei x = Anzahl der Knoten Ebene H

Der Heap hat insgesamt $\underbrace{\underbrace{2^H - 1}_{\text{Ebene 0 bis inkl. } H-1} + x}_{n}$ Knoten



Gesamtzahl der Blätter:

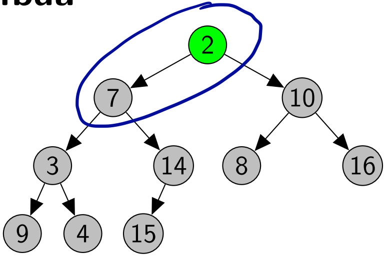
Fallunterscheid
n gerade /
n ungerade

$$x + 2^{H-1} - \left\lceil \frac{x}{2} \right\rceil = \left\lfloor x + 2^{H-1} - \frac{x}{2} \right\rfloor$$

$$= \left\lfloor 2^{H-1} + \frac{x}{2} \right\rfloor = \left\lfloor \frac{2^H + x}{2} \right\rfloor = \left\lfloor \frac{n+1}{2} \right\rfloor \overset{(*)}{=} \left\lceil \frac{n}{2} \right\rceil$$

Übersicht

Naiver Heapaufbau



kein

max Heap

Eingabe E =

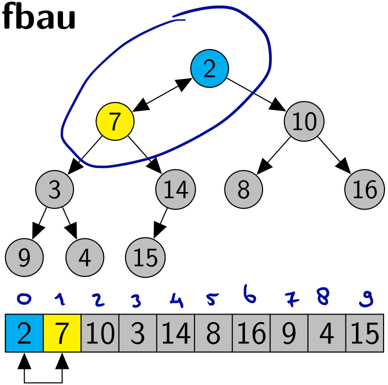
2	7	10	3	14	8	16	9	4	15
0	1	2	3	4	5	6	7	8	9

Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau

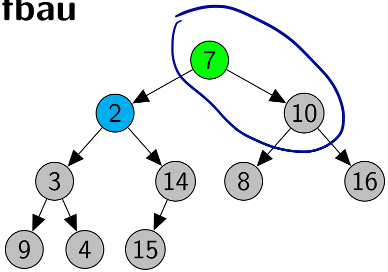


Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau



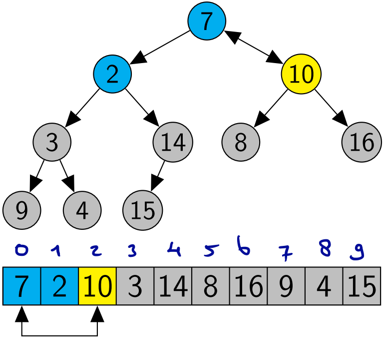
0	1	2	3	4	5	6	7	8	9
7	2	10	3	14	8	16	9	4	15

Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau

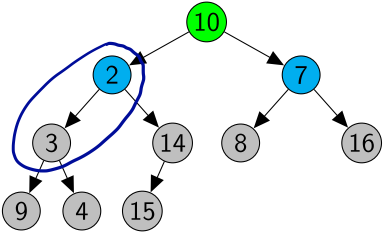


Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau



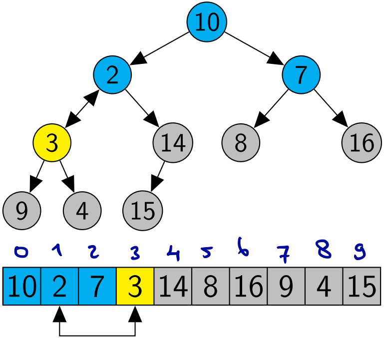
0	1	2	3	4	5	6	7	8	9
10	2	7	3	14	8	16	9	4	15

Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau

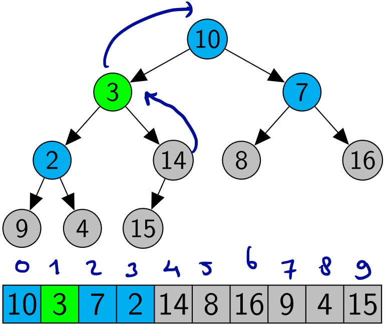


Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau

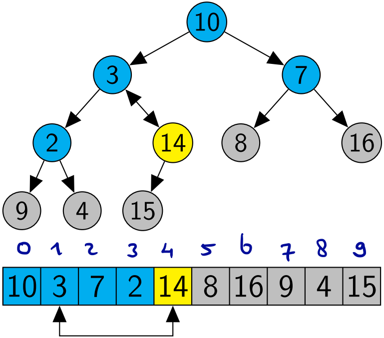


Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau

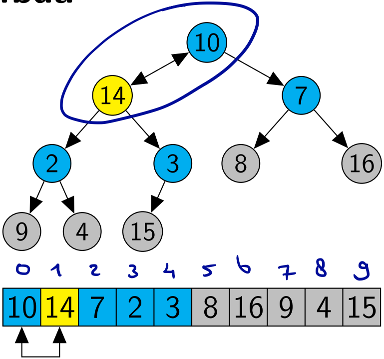


Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau

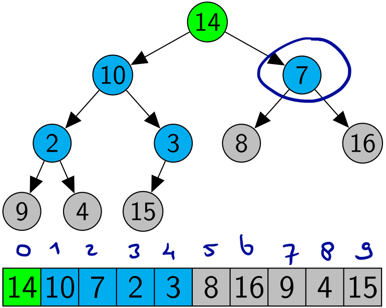


Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau

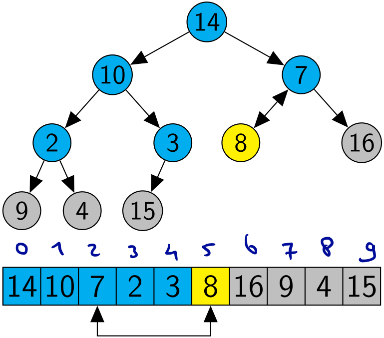


Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau

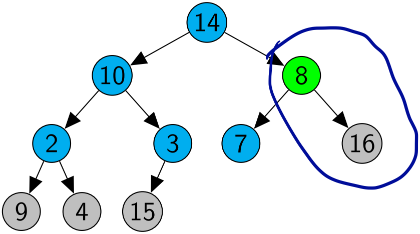


Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau



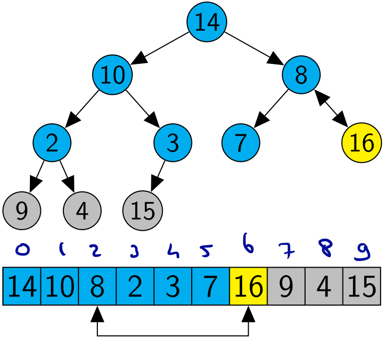
0	1	2	3	4	5	6	7	8	9
14	10	8	2	3	7	16	9	4	15

Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau

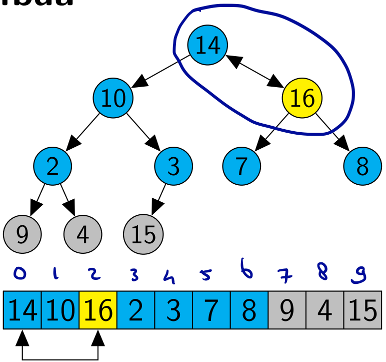


Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau

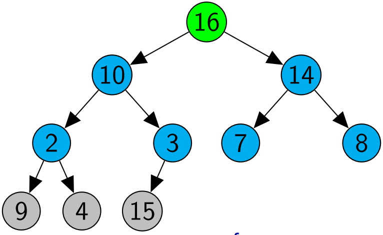


Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau



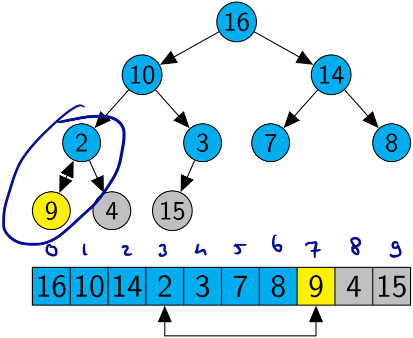
0	1	2	3	4	5	6	7	8	9
16	10	14	2	3	7	8	9	4	15

Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau

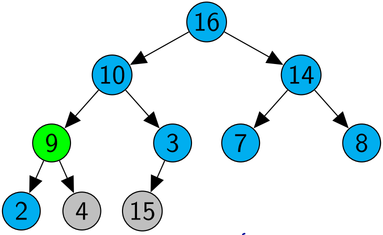


Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau



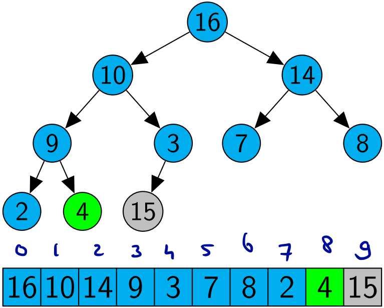
0	1	2	3	4	5	6	7	8	9
16	10	14	9	3	7	8	2	4	15

Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau

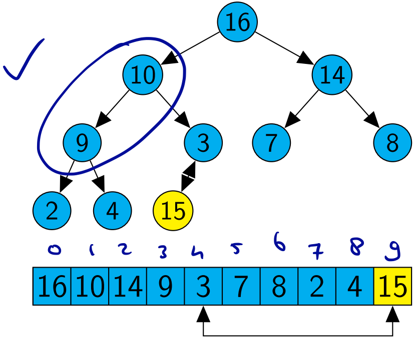


Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau

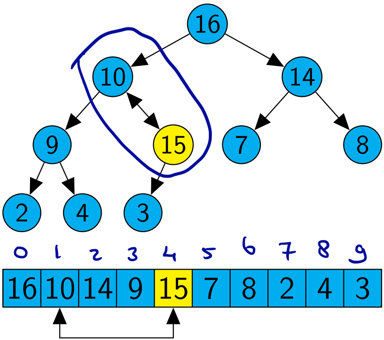


Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau



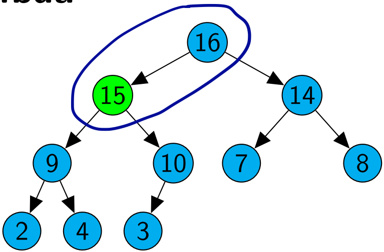
Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau

Ausgabe Heap =



Array Einbettung =

0	1	2	3	4	5	6	7	8	9
16	15	14	9	10	7	8	2	4	3

Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

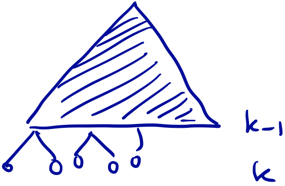
- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.

Naiver Heapaufbau – Algorithmus und Analyse

```
void bubble(int E[], int pos) {
  while (pos > 0) {
    int parent = (pos - 1) / 2;
    if (E[parent] > E[pos]) {
      break;
    }
    swap(E[parent], E[pos]);
    pos = parent;
  }
}
```

Naiver Heapaufbau – Algorithmus und Analyse

```
void bubble(int E[], int pos) {
  while (pos > 0) {
    int parent = (pos - 1) / 2;
    if (E[parent] > E[pos]) {
      break;
    }
    swap(E[parent], E[pos]);
    pos = parent;
  }
}
```



Die Höhe k eines Heaps mit n Elementen ist beschränkt durch:

$$\underline{n \leqslant 2^{k+1} - 1} \quad \Rightarrow \quad k = \underline{\lfloor \log_2 n \rfloor}$$

Naiver Heapaufbau – Algorithmus und Analyse

```
void bubble(int E[], int pos) {
  while (pos > 0) {
    int parent = (pos - 1) / 2;
    if (E[parent] > E[pos]) {
      break;
    }
    swap(E[parent], E[pos]);
    pos = parent;
  }
}
```

Die Höhe k eines Heaps mit n Elementen ist beschränkt durch:

$$n \leqslant 2^{k+1} - 1 \qquad \Rightarrow \qquad k = \lfloor \log_2 n \rfloor$$

- Damit kostet jedes Einfügen $k \approx \log_2 n$ Vergleiche.

Naiver Heapaufbau – Algorithmus und Analyse

```
void bubble(int E[], int pos) {
  while (pos > 0) {
    int parent = (pos - 1) / 2;
    if (E[parent] > E[pos]) {
      break;
    }
    swap(E[parent], E[pos]);
    pos = parent;
  }
}
```

Die Höhe k eines Heaps mit n Elementen ist beschränkt durch:

$$n \leqslant 2^{k+1} - 1 \qquad \Rightarrow \qquad k = \lfloor \log_2 n \rfloor$$

- Damit kostet jedes Einfügen $k \approx \log_2 n$ Vergleiche.

$\Rightarrow$ Zum Aufbau eines Heaps mit n Elementen benötigt man $\Theta(n \cdot \log(n))$ Vergleiche.

Naiver Heapaufbau – Algorithmus und Analyse

```
void bubble(int E[], int pos) {
  while (pos > 0) {
    int parent = (pos - 1) / 2;
    if (E[parent] > E[pos]) {
      break;
    }
    swap(E[parent], E[pos]);
    pos = parent;
  }
}
```

Die Höhe k eines Heaps mit n Elementen ist beschränkt durch:

$$n \leqslant 2^{k+1} - 1 \qquad \Rightarrow \qquad k = \lfloor \log_2 n \rfloor$$

- Damit kostet jedes Einfügen $k \approx \log_2 n$ Vergleiche.

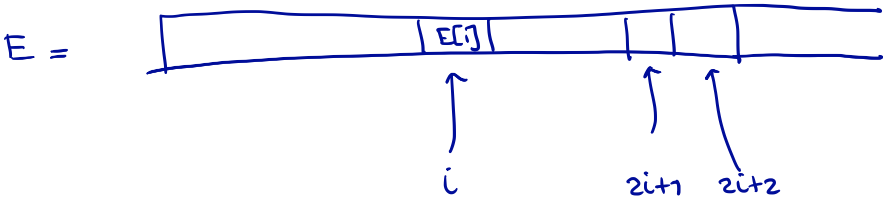
$\Rightarrow$ Zum Aufbau eines Heaps mit n Elementen benötigt man $\Theta(n \cdot \log(n))$ Vergleiche.

Es geht effizienter: `heapify` (auch: sink, fixheap) [Floyd 1964]

Heapify – Strategie

Betrachte E[i] unter der Annahme, dass der rechte und linke Teilbaum bereits ein Heap ist.

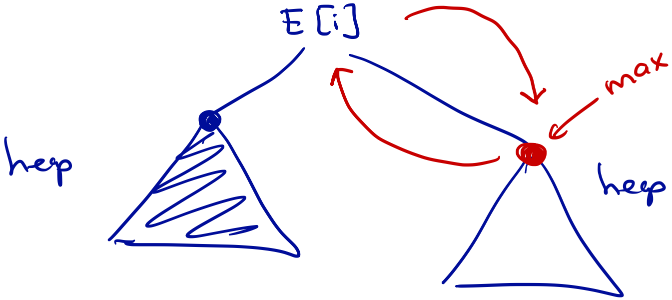
- E[i] kann kleiner als seine Kinder sein.



Heapify – Strategie

Betrachte `E[i]` unter der Annahme, dass der rechte und linke Teilbaum bereits ein Heap ist.

- `E[i]` kann kleiner als seine Kinder sein.
- Wir wollen die beiden Teilbäume – Heaps – zusammen mit `E[i]` zu einem (Gesamt-)Heap verschmelzen.



Heapify – Strategie

Betrachte `E[i]` unter der Annahme, dass der rechte und linke Teilbaum bereits ein Heap ist.

- `E[i]` kann kleiner als seine Kinder sein.
- Wir wollen die beiden Teilbäume – Heaps – zusammen mit `E[i]` zu einem (Gesamt-)Heap verschmelzen.
- Dazu lassen wir `E[i]` in den Heap hineinsinken, sodass der Teilbaum mit Wurzel `E[i]` ein Heap ist.

Heapify – Strategie

Betrachte `E[i]` unter der Annahme, dass der rechte und linke Teilbaum bereits ein Heap ist.

- `E[i]` kann kleiner als seine Kinder sein.
- Wir wollen die beiden Teilbäume – Heaps – zusammen mit `E[i]` zu einem (Gesamt-)Heap verschmelzen.
- Dazu lassen wir `E[i]` in den Heap hineinsinken, sodass der Teilbaum mit Wurzel `E[i]` ein Heap ist.

Heapify

- Finde das Maximum der Werte `E[i]` und seiner Kinder.

Heapify – Strategie

Betrachte `E[i]` unter der Annahme, dass der rechte und linke Teilbaum bereits ein Heap ist.

- `E[i]` kann kleiner als seine Kinder sein.
- Wir wollen die beiden Teilbäume – Heaps – zusammen mit `E[i]` zu einem (Gesamt-)Heap verschmelzen.
- Dazu lassen wir `E[i]` in den Heap hineinsinken, sodass der Teilbaum mit Wurzel `E[i]` ein Heap ist.

Heapify

- Finde das Maximum der Werte `E[i]` und seiner Kinder.
- Ist `E[i]` bereits das größte Element, dann ist dieser gesamte Teilbaum auch ein Heap. Fertig.

Heapify – Strategie

Betrachte `E[i]` unter der Annahme, dass der rechte und linke Teilbaum bereits ein Heap ist.

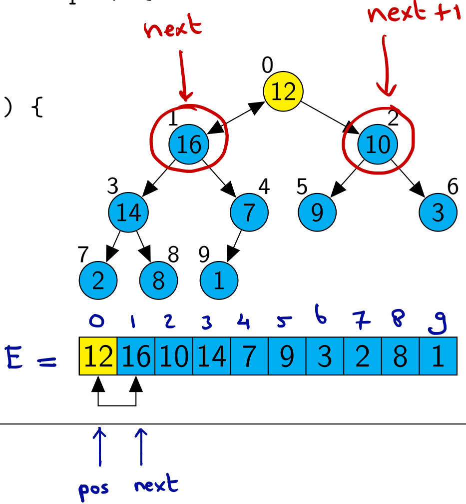
- `E[i]` kann kleiner als seine Kinder sein.
- Wir wollen die beiden Teilbäume – Heaps – zusammen mit `E[i]` zu einem (Gesamt-)Heap verschmelzen.
- Dazu lassen wir `E[i]` in den Heap hineinsinken, sodass der Teilbaum mit Wurzel `E[i]` ein Heap ist.

Heapify

- Finde das Maximum der Werte `E[i]` und seiner Kinder.
- Ist `E[i]` bereits das größte Element, dann ist dieser gesamte Teilbaum auch ein Heap. Fertig.
- Andernfalls tausche `E[i]` mit dem größten Element und führe Heapify in diesem Unterbaum weiter aus.

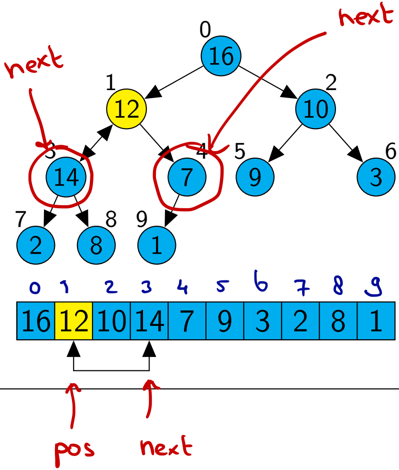
Heapify – Algorithmus und Beispiel

```
void heapify(int E[], int n, int pos) {
  int next = 2 * pos + 1;
  while (next < n) {
    if (next + 1 < n &&
        E[next + 1] > E[next]) {
      next = next + 1;
    }
    if (E[pos] >= E[next]) {
      break;
    }
    swap(E[pos], E[next]);
    pos = next;
    next = 2 * pos + 1;
  }
}
```



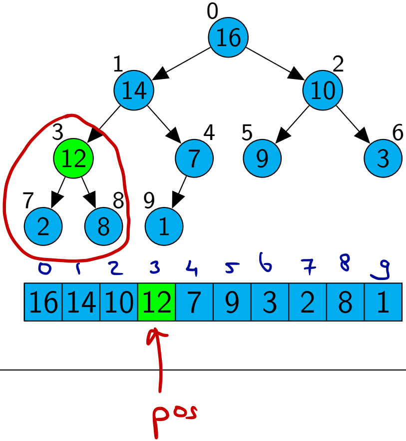
Heapify – Algorithmus und Beispiel

```
void heapify(int E[], int n, int pos) {
  int next = 2 * pos + 1;
  while (next < n) {
    if (next + 1 < n &&
        E[next + 1] > E[next]) {
      next = next + 1;
    }
    if (E[pos] >= E[next]) {
      break;
    }
    swap(E[pos], E[next]);
    pos = next;
    next = 2 * pos + 1;
  }
}
```



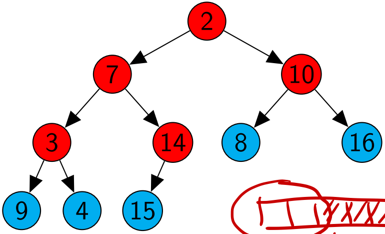
Heapify – Algorithmus und Beispiel

```
void heapify(int E[], int n, int pos) {
  int next = 2 * pos + 1;
  while (next < n) {
    if (next + 1 < n &&
        E[next + 1] > E[next]) {
      next = next + 1;
    }
    if (E[pos] >= E[next]) {
      break;
    }
    swap(E[pos], E[next]);
    pos = next;
    next = 2 * pos + 1;
  }
}
```



Heapaufbau – Algorithmus und Beispiel

Strategie: Wandle das Array von unten nach oben (bottom-up) in einen Heap um.



Blätter

i

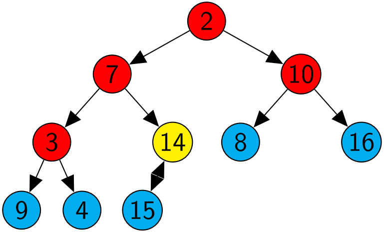
```
void buildHeap(int E[]) {
  for (int i = E.length / 2 - 1; i >= 0; i--) {
    heapify(E, E.length, i);
  }
}
```

Wurzeln von Heaps

Schleifeninvariant: Nach jedem Aufruf von `heapify(E, E.length, i)` sind die Knoten `i, ..., E.length - 1` schon Wurzeln von Heaps.

Heapaufbau – Algorithmus und Beispiel

Strategie: Wandle das Array von unten nach oben (bottom-up) in einen Heap um.

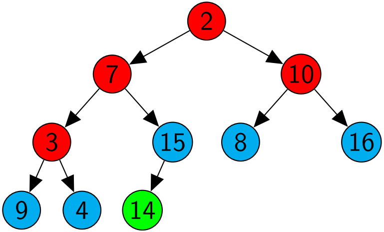



```
void buildHeap(int E[]) {
  for (int i = E.length / 2 - 1; i >= 0; i--) {
    heapify(E, E.length, i);
  }
}
```

Schleifeninvariant: Nach jedem Aufruf von `heapify(E, E.length, i)` sind die Knoten `i, ..., E.length - 1` schon Wurzeln von Heaps.

Heapaufbau – Algorithmus und Beispiel

Strategie: Wandle das Array von unten nach oben (bottom-up) in einen Heap um.

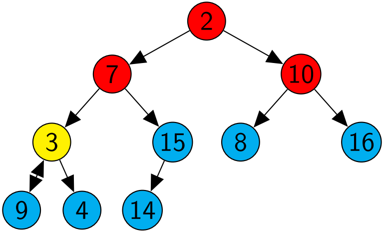



```
void buildHeap(int E[]) {
  for (int i = E.length / 2 - 1; i >= 0; i--) {
    heapify(E, E.length, i);
  }
}
```

Schleifeninvariant: Nach jedem Aufruf von `heapify(E, E.length, i)` sind die Knoten `i, ..., E.length - 1` schon Wurzeln von Heaps.

Heapaufbau – Algorithmus und Beispiel

Strategie: Wandle das Array von unten nach oben (bottom-up) in einen Heap um.

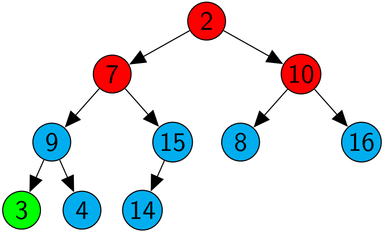



```
void buildHeap(int E[]) {
  for (int i = E.length / 2 - 1; i >= 0; i--) {
    heapify(E, E.length, i);
  }
}
```

Schleifeninvariant: Nach jedem Aufruf von `heapify(E, E.length, i)` sind die Knoten `i, ..., E.length - 1` schon Wurzeln von Heaps.

Heapaufbau – Algorithmus und Beispiel

Strategie: Wandle das Array von unten nach oben (bottom-up) in einen Heap um.

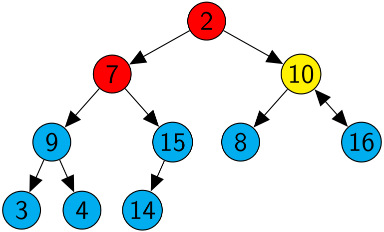



```
void buildHeap(int E[]) {
  for (int i = E.length / 2 - 1; i >= 0; i--) {
    heapify(E, E.length, i);
  }
}
```

Schleifeninvariant: Nach jedem Aufruf von `heapify(E, E.length, i)` sind die Knoten `i, ..., E.length - 1` schon Wurzeln von Heaps.

Heapaufbau – Algorithmus und Beispiel

Strategie: Wandle das Array von unten nach oben (bottom-up) in einen Heap um.

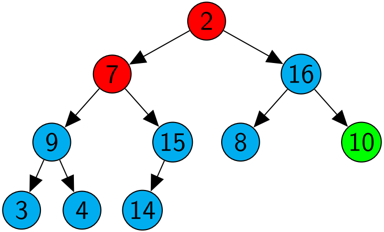



```
void buildHeap(int E[]) {
  for (int i = E.length / 2 - 1; i >= 0; i--) {
    heapify(E, E.length, i);
  }
}
```

Schleifeninvariant: Nach jedem Aufruf von `heapify(E, E.length, i)` sind die Knoten `i, ..., E.length - 1` schon Wurzeln von Heaps.

Heapaufbau – Algorithmus und Beispiel

Strategie: Wandle das Array von unten nach oben (bottom-up) in einen Heap um.

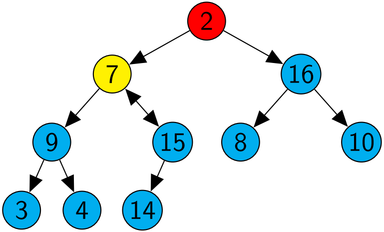



```
void buildHeap(int E[]) {
  for (int i = E.length / 2 - 1; i >= 0; i--) {
    heapify(E, E.length, i);
  }
}
```

Schleifeninvariant: Nach jedem Aufruf von `heapify(E, E.length, i)` sind die Knoten `i, ..., E.length - 1` schon Wurzeln von Heaps.

Heapaufbau – Algorithmus und Beispiel

Strategie: Wandle das Array von unten nach oben (bottom-up) in einen Heap um.

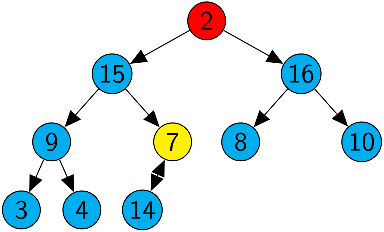



```
void buildHeap(int E[]) {
  for (int i = E.length / 2 - 1; i >= 0; i--) {
    heapify(E, E.length, i);
  }
}
```

Schleifeninvariant: Nach jedem Aufruf von `heapify(E, E.length, i)` sind die Knoten `i, ..., E.length - 1` schon Wurzeln von Heaps.

Heapaufbau – Algorithmus und Beispiel

Strategie: Wandle das Array von unten nach oben (bottom-up) in einen Heap um.

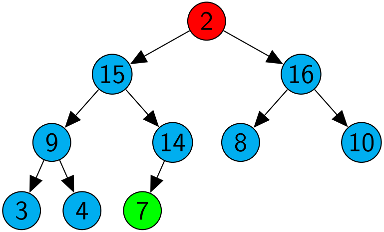



```
void buildHeap(int E[]) {
  for (int i = E.length / 2 - 1; i >= 0; i--) {
    heapify(E, E.length, i);
  }
}
```

Schleifeninvariant: Nach jedem Aufruf von `heapify(E, E.length, i)` sind die Knoten `i, ..., E.length - 1` schon Wurzeln von Heaps.

Heapaufbau – Algorithmus und Beispiel

Strategie: Wandle das Array von unten nach oben (bottom-up) in einen Heap um.

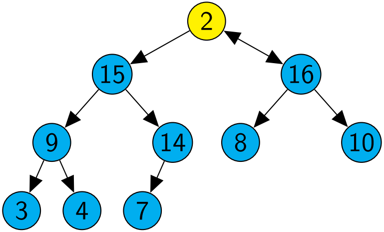



```
void buildHeap(int E[]) {
  for (int i = E.length / 2 - 1; i >= 0; i--) {
    heapify(E, E.length, i);
  }
}
```

Schleifeninvariant: Nach jedem Aufruf von `heapify(E, E.length, i)` sind die Knoten `i, ..., E.length - 1` schon Wurzeln von Heaps.

Heapaufbau – Algorithmus und Beispiel

Strategie: Wandle das Array von unten nach oben (bottom-up) in einen Heap um.

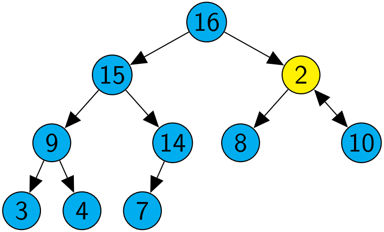



```
void buildHeap(int E[]) {
  for (int i = E.length / 2 - 1; i >= 0; i--) {
    heapify(E, E.length, i);
  }
}
```

Schleifeninvariant: Nach jedem Aufruf von `heapify(E, E.length, i)` sind die Knoten `i, ..., E.length - 1` schon Wurzeln von Heaps.

Heapaufbau – Algorithmus und Beispiel

Strategie: Wandle das Array von unten nach oben (bottom-up) in einen Heap um.

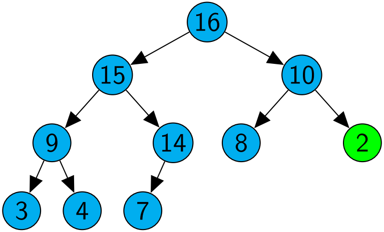



```
void buildHeap(int E[]) {
  for (int i = E.length / 2 - 1; i >= 0; i--) {
    heapify(E, E.length, i);
  }
}
```

Schleifeninvariant: Nach jedem Aufruf von `heapify(E, E.length, i)` sind die Knoten `i, ..., E.length - 1` schon Wurzeln von Heaps.

Heapaufbau – Algorithmus und Beispiel

Strategie: Wandle das Array von unten nach oben (bottom-up) in einen Heap um.




```
void buildHeap(int E[]) {
  for (int i = E.length / 2 - 1; i >= 0; i--) {
    heapify(E, E.length, i);
  }
}
```

Schleifeninvariant: Nach jedem Aufruf von `heapify(E, E.length, i)` sind die Knoten `i, ..., E.length - 1` schon Wurzeln von Heaps.

Konstruktion eines Heaps

Lemma

Der Algorithmus `buildHeap` *ist korrekt und terminiert.*

Konstruktion eines Heaps

Lemma

Der Algorithmus `buildHeap` *ist korrekt und terminiert.*

- Initialisierung: Jeder Knoten $i = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \ldots$ ist ein Blatt und damit Wurzel eines trivialen Heaps.

E =

Blätter

= Wurzel eines Heaps

Konstruktion eines Heaps

Lemma

Der Algorithmus `buildHeap` *ist korrekt und terminiert.*

- Initialisierung: Jeder Knoten $i = \lfloor n/2 \rfloor, \ \lfloor n/2 \rfloor + 1, \ldots$ ist ein Blatt und damit Wurzel eines trivialen Heaps.
- Schleifeninvariante: Zu Beginn der **for**-Schleife ist jeder Knoten $i{+}1, \ldots, E.length$ die Wurzel eines Heaps.

Konstruktion eines Heaps

Lemma

Der Algorithmus `buildHeap` *ist korrekt und terminiert.*

- Initialisierung: Jeder Knoten $i = \lfloor n/2 \rfloor,\ \lfloor n/2 \rfloor + 1, \ldots$ ist ein Blatt und damit Wurzel eines trivialen Heaps.
- Schleifeninvariante: Zu Beginn der **for**-Schleife ist jeder Knoten $i{+}1, \ldots, E.length$ die Wurzel eines Heaps.
- In jeder Iteration sind alle Kinder des Knotens i bereits Wurzeln eines Heaps (Schleifeninvariante).

Konstruktion eines Heaps

Lemma

Der Algorithmus `buildHeap` *ist korrekt und terminiert.*

- Initialisierung: Jeder Knoten $i = \lfloor n/2 \rfloor, \; \lfloor n/2 \rfloor + 1, \ldots$ ist ein Blatt und damit Wurzel eines trivialen Heaps.
- Schleifeninvariante: Zu Beginn der **for**-Schleife ist jeder Knoten $i{+}1, \ldots, E.length$ die Wurzel eines Heaps.
- In jeder Iteration sind alle Kinder des Knotens i bereits Wurzeln eines Heaps (Schleifeninvariante).

$\Rightarrow$ Bedingung für den Aufruf von `heapify` ist erfüllt.

- Dekrementierung von i stellt Schleifeninvariante wieder her.

Konstruktion eines Heaps

Lemma

Der Algorithmus `buildHeap` *ist korrekt und terminiert.*

- Initialisierung: Jeder Knoten $i = \lfloor n/2 \rfloor, \ \lfloor n/2 \rfloor + 1, \ldots$ ist ein Blatt und damit Wurzel eines trivialen Heaps.
- Schleifeninvariante: Zu Beginn der **for**-Schleife ist jeder Knoten $i{+}1, \ldots, E.length$ die Wurzel eines Heaps.
- In jeder Iteration sind alle Kinder des Knotens i bereits Wurzeln eines Heaps (Schleifeninvariante).

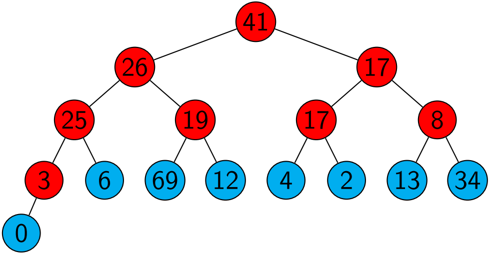
$\Rightarrow$ Bedingung für den Aufruf von `heapify` ist erfüllt.

- Dekrementierung von i stellt Schleifeninvariante wieder her.
- Terminierung: Bei $i = 0$ ist gemäß Schleifeninvariante jeder Knoten $1, 2, \ldots, n$ die Wurzel eines Heaps.

Übersicht

 Heapsort

Heapsort – Algorithmus und Beispiel

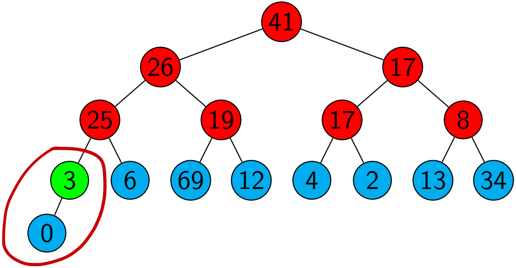


Eingabe E =

41	26	17	25	19	17	8	3	6	69	12	4	2	13	34	0

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

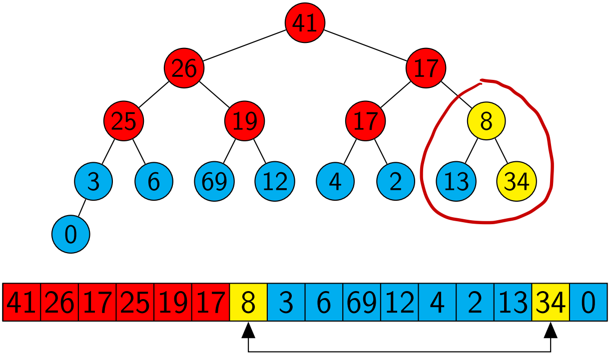
Heapsort – Algorithmus und Beispiel



41	26	17	25	19	17	8	3	6	69	12	4	2	13	34	0

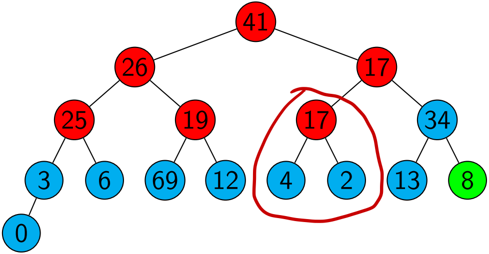
```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel

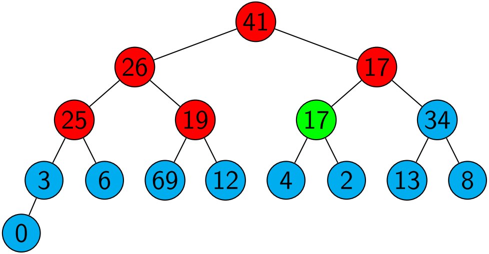


41	26	17	25	19	17	34	3	6	69	12	4	2	13	8	0

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

 Heapsort

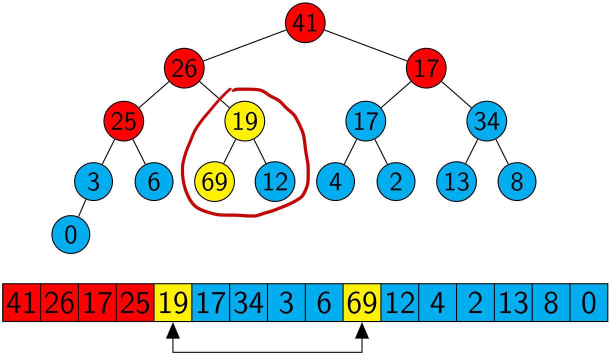
Heapsort – Algorithmus und Beispiel



41	26	17	25	19	17	34	3	6	69	12	4	2	13	8	0

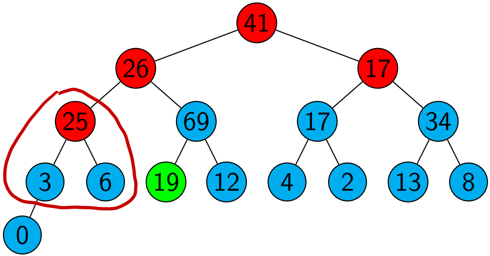
```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

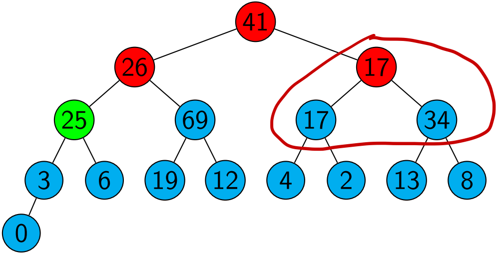
Heapsort – Algorithmus und Beispiel



41	26	17	25	69	17	34	3	6	19	12	4	2	13	8	0

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

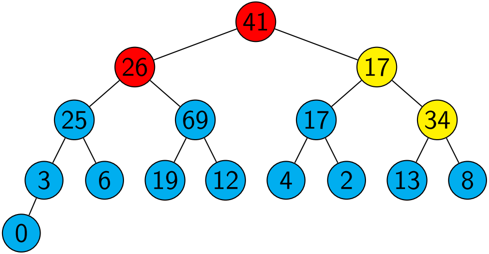
Heapsort – Algorithmus und Beispiel



41	26	17	25	69	17	34	3	6	19	12	4	2	13	8	0

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

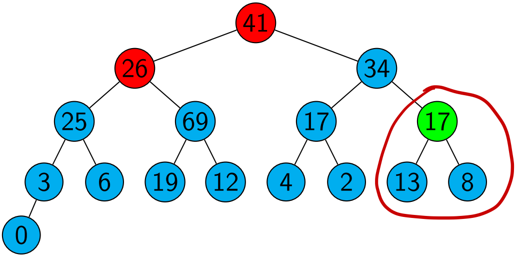
Heapsort – Algorithmus und Beispiel



41	26	17	25	69	17	34	3	6	19	12	4	2	13	8	0

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

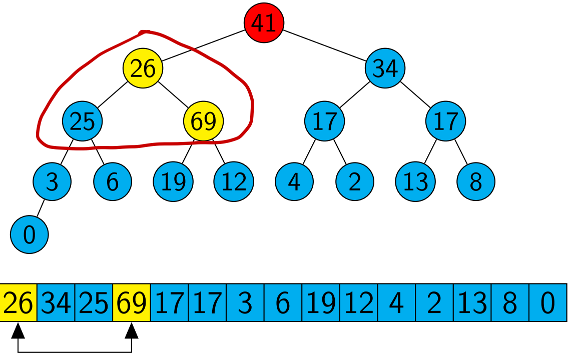
Heapsort – Algorithmus und Beispiel



41	26	34	25	69	17	17	3	6	19	12	4	2	13	8	0

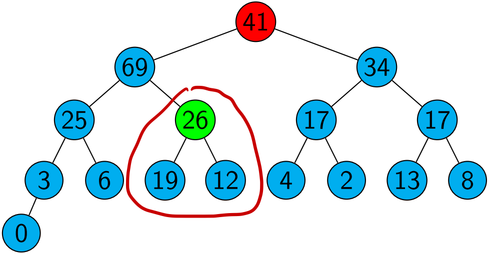
```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

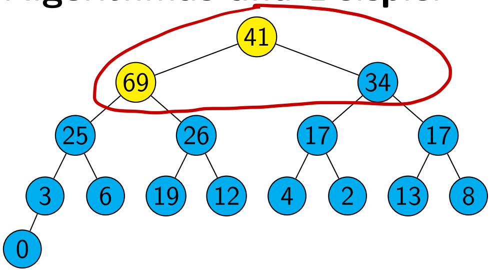
Heapsort – Algorithmus und Beispiel



41	69	34	25	26	17	17	3	6	19	12	4	2	13	8	0

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

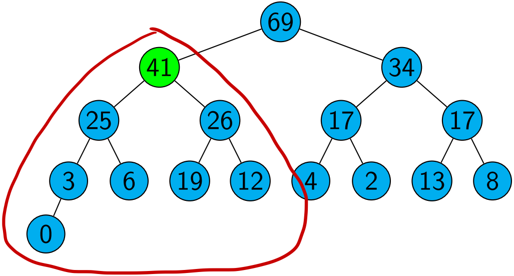
Heapsort – Algorithmus und Beispiel



41	69	34	25	26	17	17	3	6	19	12	4	2	13	8	0

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel

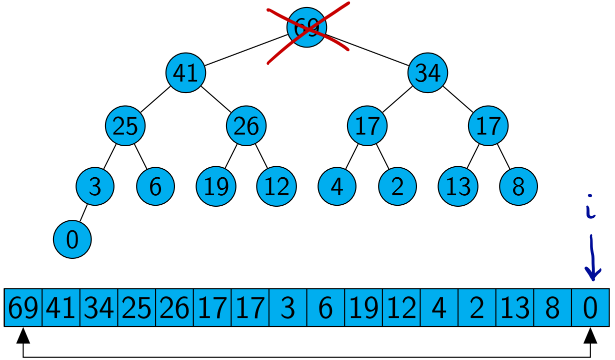


Ausgabe buildHeap

69	41	34	25	26	17	17	3	6	19	12	4	2	13	8	0

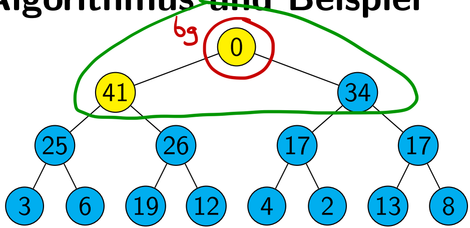
```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

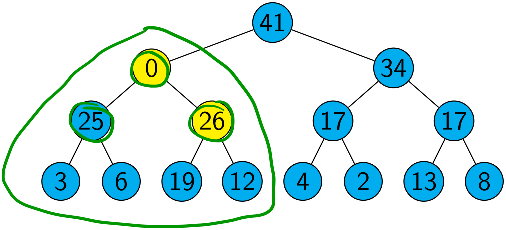
Heapsort – Algorithmus und Beispiel



0	41	34	25	26	17	17	3	6	19	12	4	2	13	8	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

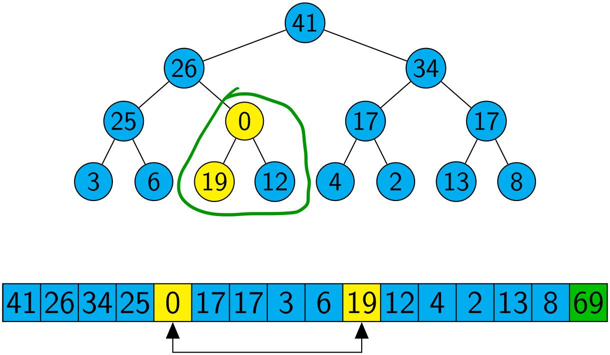
Heapsort – Algorithmus und Beispiel



41	0	34	25	26	17	17	3	6	19	12	4	2	13	8	69

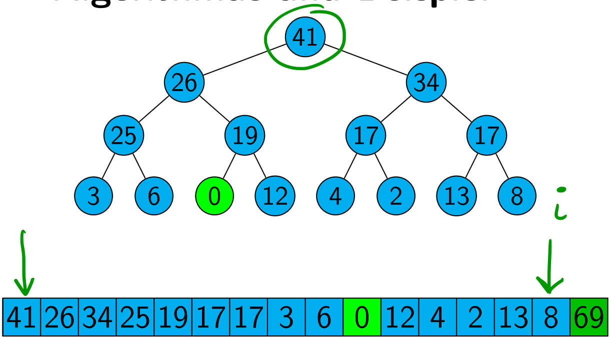
```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel



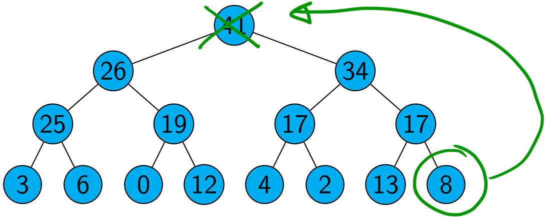

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

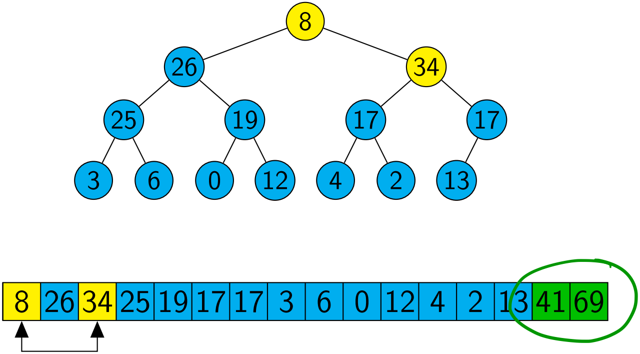
Heapsort – Algorithmus und Beispiel



41	26	34	25	19	17	17	3	6	0	12	4	2	13	8	69

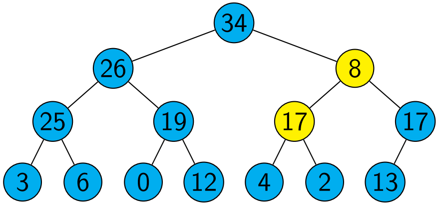
```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

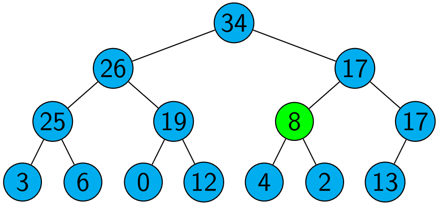
Heapsort – Algorithmus und Beispiel



34	26	8	25	19	17	17	3	6	0	12	4	2	13	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

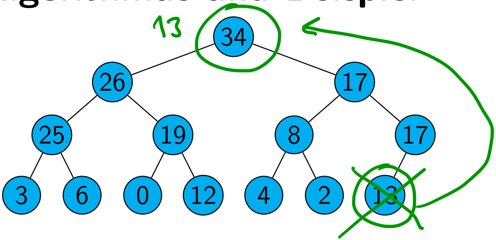
Heapsort – Algorithmus und Beispiel



34	26	17	25	19	8	17	3	6	0	12	4	2	13	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

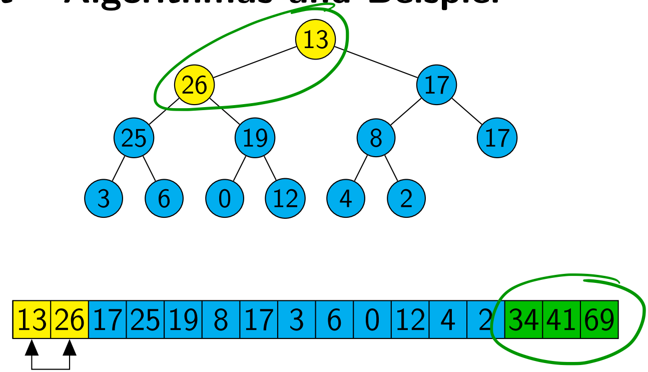
Heapsort – Algorithmus und Beispiel



34	26	17	25	19	8	17	3	6	0	12	4	2	13	41	69

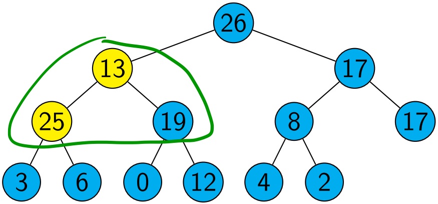
```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

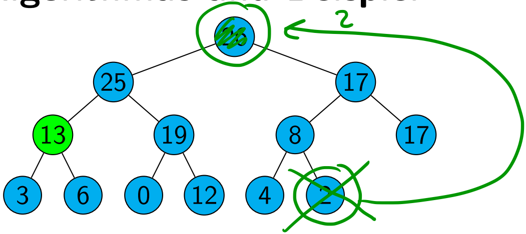
Heapsort – Algorithmus und Beispiel



26	13	17	25	19	8	17	3	6	0	12	4	2	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

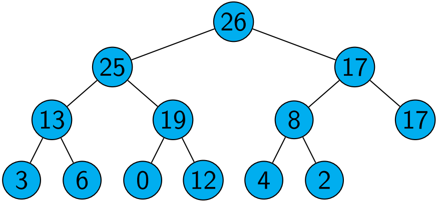
Heapsort – Algorithmus und Beispiel

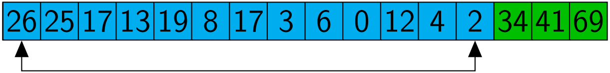


26	25	17	13	19	8	17	3	6	0	12	4	2	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

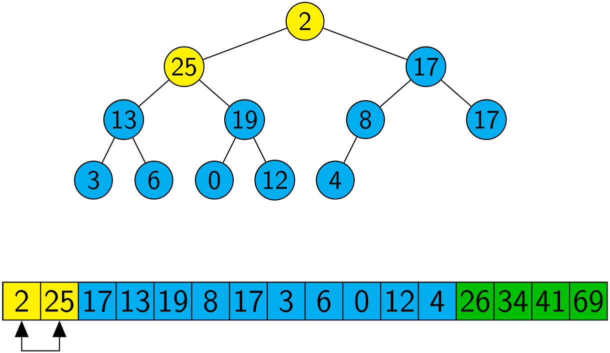
Heapsort – Algorithmus und Beispiel





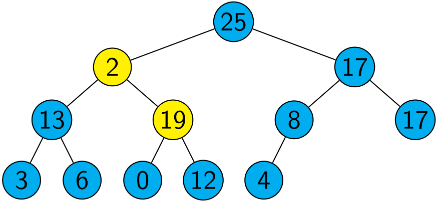

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

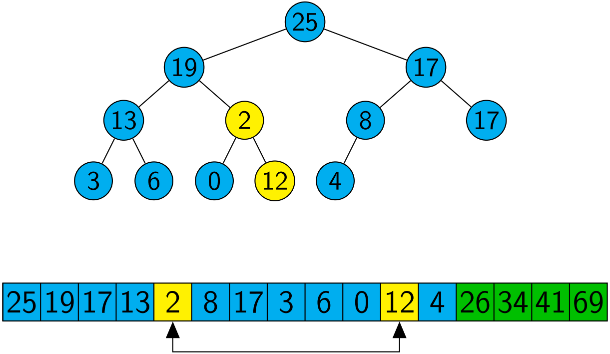
Heapsort – Algorithmus und Beispiel



25	2	17	13	19	8	17	3	6	0	12	4	26	34	41	69

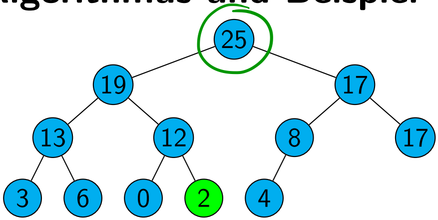
```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

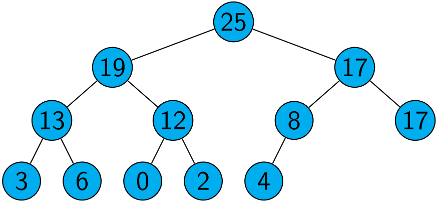
Heapsort – Algorithmus und Beispiel



25	19	17	13	12	8	17	3	6	0	2	4	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

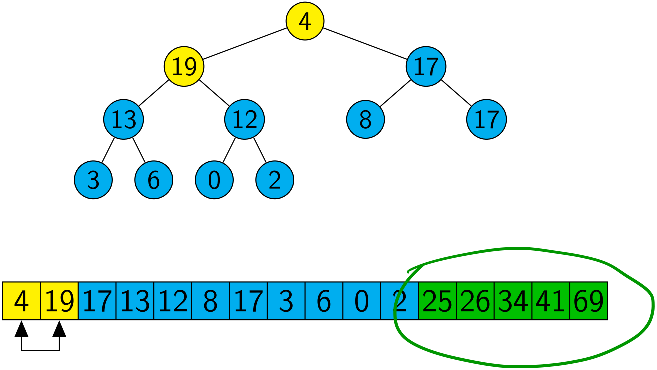
Heapsort – Algorithmus und Beispiel



25	19	17	13	12	8	17	3	6	0	2	4	26	34	41	69

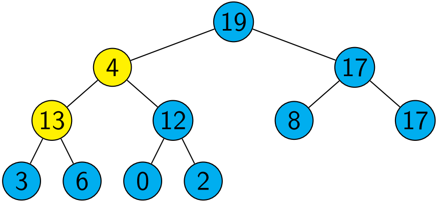
```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```java
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

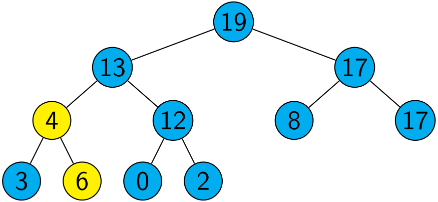
Heapsort – Algorithmus und Beispiel



19	4	17	13	12	8	17	3	6	0	2	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

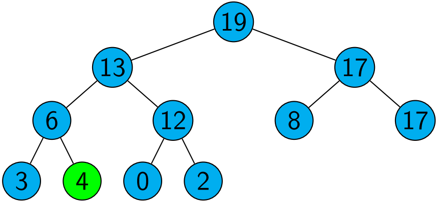
Heapsort – Algorithmus und Beispiel



19	13	17	4	12	8	17	3	6	0	2	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

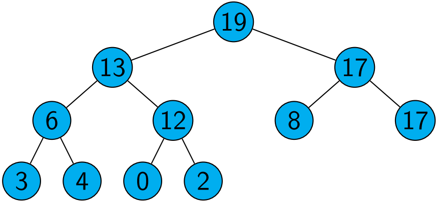
Heapsort – Algorithmus und Beispiel



19	13	17	6	12	8	17	3	4	0	2	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

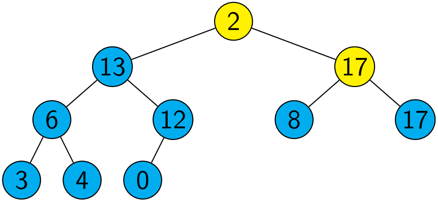
Heapsort – Algorithmus und Beispiel



19	13	17	6	12	8	17	3	4	0	2	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

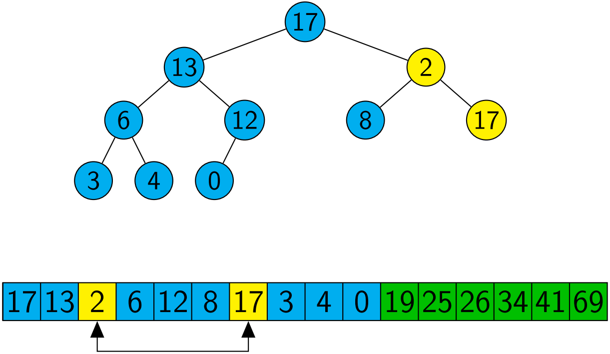
Heapsort – Algorithmus und Beispiel



2	13	17	6	12	8	17	3	4	0	19	25	26	34	41	69

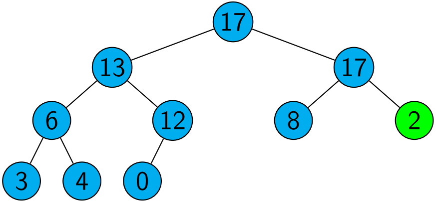
```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

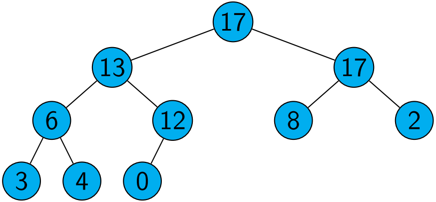
Heapsort – Algorithmus und Beispiel



17	13	17	6	12	8	2	3	4	0	19	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

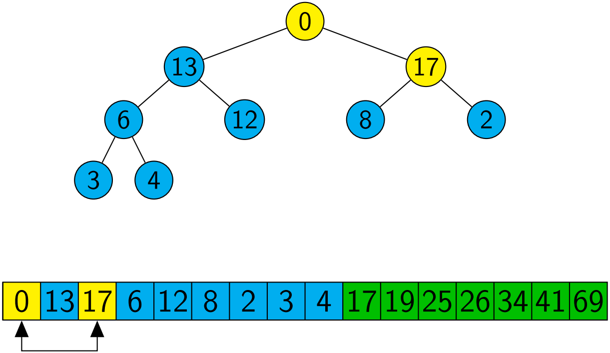
Heapsort – Algorithmus und Beispiel



17	13	17	6	12	8	2	3	4	0	19	25	26	34	41	69

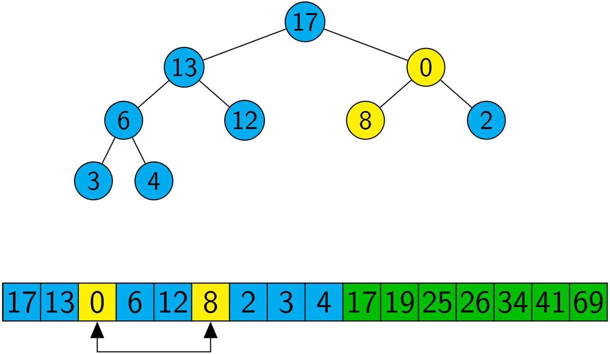
```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel



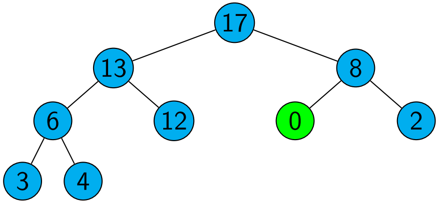

```java
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

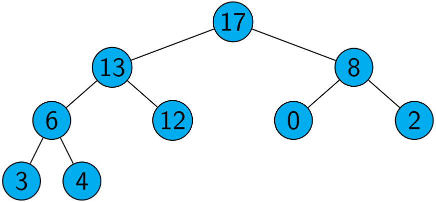
Heapsort – Algorithmus und Beispiel



17	13	8	6	12	0	2	3	4	17	19	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

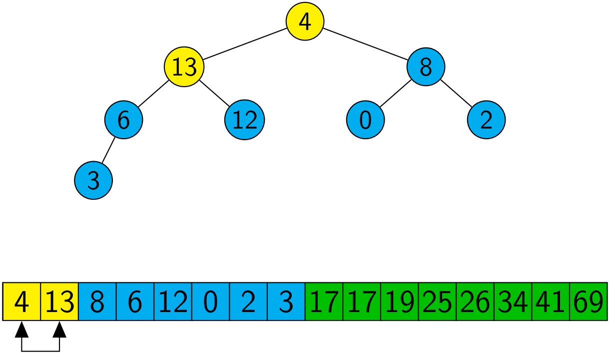
Heapsort – Algorithmus und Beispiel



17	13	8	6	12	0	2	3	4	17	19	25	26	34	41	69

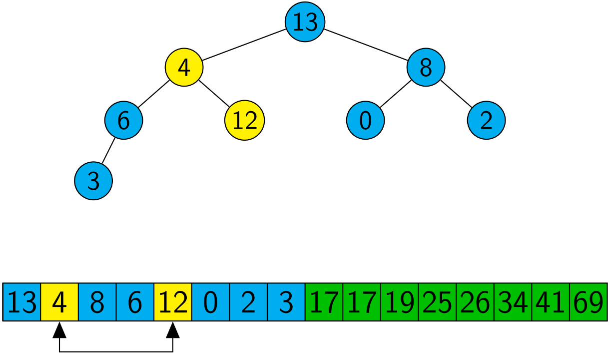
```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel



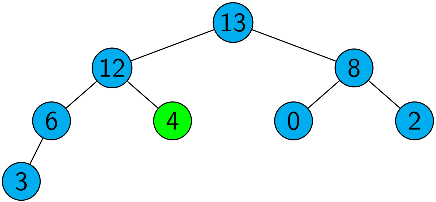

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

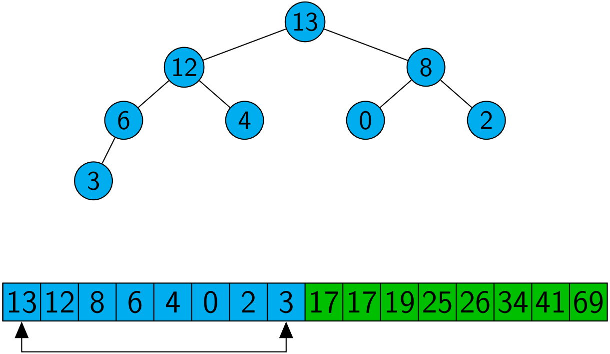
Heapsort – Algorithmus und Beispiel



13	12	8	6	4	0	2	3	17	17	19	25	26	34	41	69

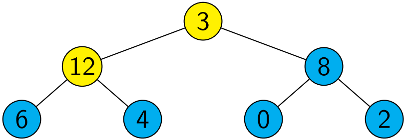
```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

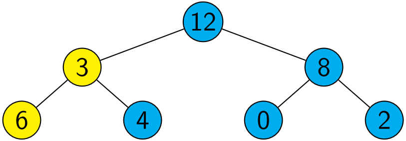
Heapsort – Algorithmus und Beispiel

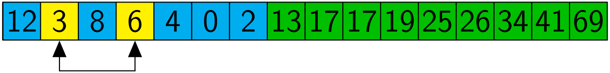


3	12	8	6	4	0	2	13	17	17	19	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

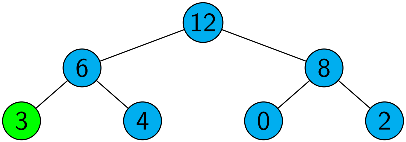
Heapsort – Algorithmus und Beispiel






```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

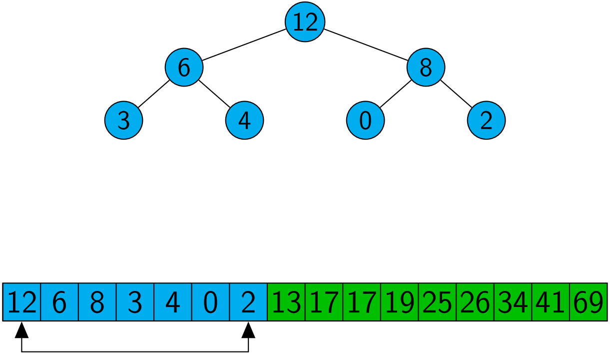
Heapsort – Algorithmus und Beispiel



12	6	8	3	4	0	2	13	17	17	19	25	26	34	41	69

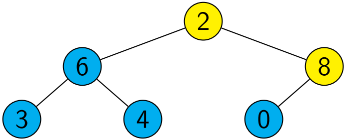
```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

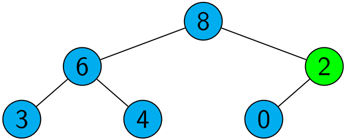
Heapsort – Algorithmus und Beispiel



2	6	8	3	4	0	12	13	17	17	19	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

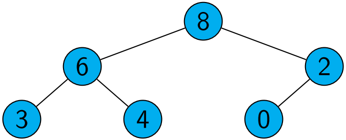
Heapsort – Algorithmus und Beispiel



8	6	2	3	4	0	12	13	17	17	19	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

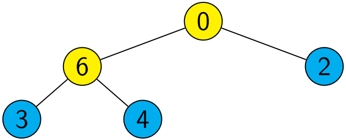
Heapsort – Algorithmus und Beispiel



8	6	2	3	4	0	12	13	17	17	19	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel

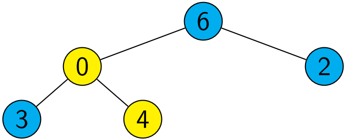


0	6	2	3	4	8	12	13	17	17	19	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

 Heapsort

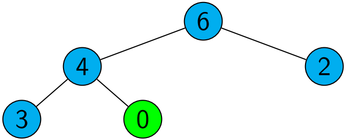
Heapsort – Algorithmus und Beispiel



6	0	2	3	4	8	12	13	17	17	19	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

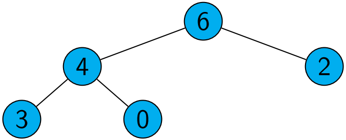
Heapsort – Algorithmus und Beispiel



6	4	2	3	0	8	12	13	17	17	19	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

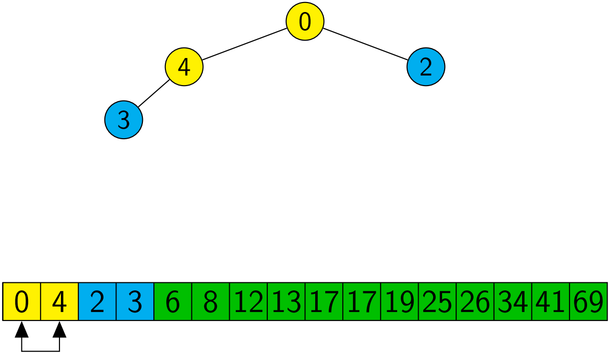
Heapsort – Algorithmus und Beispiel



6	4	2	3	0	8	12	13	17	17	19	25	26	34	41	69

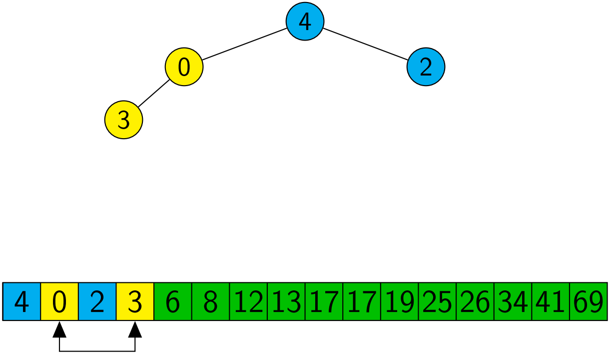
```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel



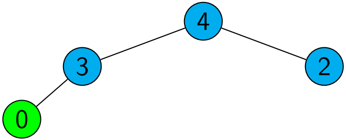

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

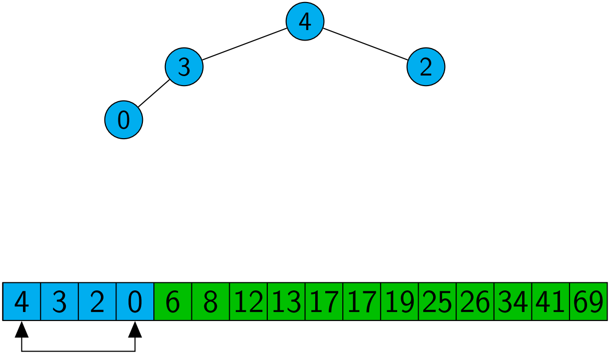
Heapsort – Algorithmus und Beispiel



4	3	2	0	6	8	12	13	17	17	19	25	26	34	41	69

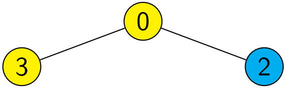
```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

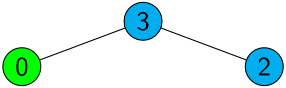
Heapsort – Algorithmus und Beispiel



0	3	2	4	6	8	12	13	17	17	19	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

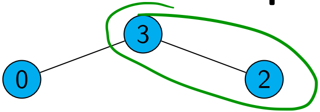
Heapsort – Algorithmus und Beispiel



3	0	2	4	6	8	12	13	17	17	19	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

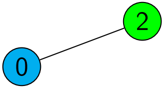
Heapsort – Algorithmus und Beispiel



3	0	2	4	6	8	12	13	17	17	19	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

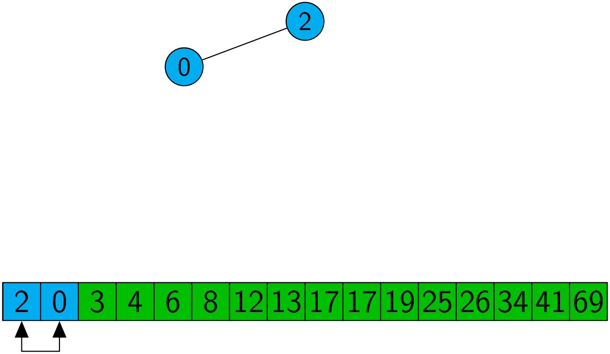
Heapsort – Algorithmus und Beispiel



2	0	3	4	6	8	12	13	17	17	19	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel




```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

Heapsort – Algorithmus und Beispiel

0	2	3	4	6	8	12	13	17	17	19	25	26	34	41	69

```
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    heapify(E, i, 0);
  }
}
```

beendet

Heapsort – Analyse

- Die Worst-Case Komplexität von *Heapify* ist maximal $2 \cdot \lfloor \log_2 n \rfloor$ für n Knoten.

Heap konstruieren (buildHeap)

beim Sortieren in der for-Schleife

Heapsort – Analyse

- Die Worst-Case Komplexität von *Heapify* ist ~~maximal~~ $2 \cdot \lfloor \log_2 n \rfloor$ für n Knoten.
  - für einen Heap mit $\underbrace{\text{Level}}_{\text{Höhe}}$ k, gibt es $2 \cdot k$ Vergleiche im Worst-Case

Heapsort – Analyse

- Die Worst-Case Komplexität von *Heapify* ist maximal $2 \cdot \lfloor \log_2 n \rfloor$ für n Knoten.
  - für einen Heap mit Level k, gibt es $2 \cdot k$ Vergleiche im Worst-Case
- Die Worst-Case Komplexität von *buildHeap* ist $\Theta(n)$

(Beweis: spätere Folie)

→ Folie 18

$\Theta(n) \leftarrow$ buildheap(E);

n−1 Iterationen

```
for ( . . . . . ) {
    . . . .
    heapify ( . . . )
}
```

Heapsort – Analyse

- Die Worst-Case Komplexität von *Heapify* ist maximal $2 \cdot \lfloor \log_2 n \rfloor$ für n Knoten.
  - für einen Heap mit Level k, gibt es $2 \cdot k$ Vergleiche im Worst-Case
- Die Worst-Case Komplexität von *buildHeap* ist $\Theta(n)$

  (Beweis: spätere Folie)
- Für Heapsort erhalten wir somit:

$$W(n) \quad = \quad \underbrace{\left(\sum_{i=1}^{n-1} 2 \cdot \lfloor \log_2 i \rfloor \right)}_{\text{for-Schleife}} + n$$

buildheap

Heapsort – Analyse

- Die Worst-Case Komplexität von *Heapify* ist maximal $2 \cdot \lfloor \log_2 n \rfloor$ für n Knoten.
  - für einen Heap mit Level k, gibt es $2 \cdot k$ Vergleiche im Worst-Case
- Die Worst-Case Komplexität von *buildHeap* ist $\Theta(n)$

  (Beweis: spätere Folie)
- Für Heapsort erhalten wir somit:

$$
\begin{aligned}
W(n) &= \left(\sum_{i=1}^{n-1} 2 \cdot \lfloor \log_2 i \rfloor \right) + n \\
&\leqslant 2 \cdot \left(\sum_{i=1}^{n-1} \log_2 n \right) + n
\end{aligned}
$$

Heapsort – Analyse

- Die Worst-Case Komplexität von *Heapify* ist maximal $2 \cdot \lfloor \log_2 n \rfloor$ für n Knoten.
  - für einen Heap mit Level k, gibt es $2 \cdot k$ Vergleiche im Worst-Case
- Die Worst-Case Komplexität von *buildHeap* ist $\Theta(n)$

  (Beweis: spätere Folie)
- Für Heapsort erhalten wir somit:

$$
\begin{aligned}
W(n) &= \left(\sum_{i=1}^{n-1} 2 \cdot \lfloor \log_2 i \rfloor \right) + n \\
&\leqslant 2 \left(\sum_{i=1}^{n-1} \log_2 n \right) + n \\
&= n + 2 \cdot \underbrace{(n-1) \cdot \log_2 n}
\end{aligned}
$$

$$\underbrace{\log n + \ldots + \log n}_{n-1 \text{ Mal}}$$

Heapsort – Analyse

- Die Worst-Case Komplexität von *Heapify* ist maximal $2 \cdot \lfloor \log_2 n \rfloor$ für n Knoten.
  - für einen Heap mit Level k, gibt es $2 \cdot k$ Vergleiche im Worst-Case
- Die Worst-Case Komplexität von *buildHeap* ist $\Theta(n)$

  (Beweis: spätere Folie)
- Für Heapsort erhalten wir somit:

$$
\begin{aligned}
W(n) &= \left(\sum_{i=1}^{n-1} 2 \cdot \lfloor \log_2 i \rfloor \right) + n \\
&\leqslant 2 \cdot \left(\sum_{i=1}^{n-1} \log_2 n \right) + n \\
&= n + 2 \cdot (n-1) \cdot \log_2 n
\end{aligned}
$$

$$\Rightarrow \boxed{W(n) \in O(n \cdot \log n)}$$

optimal

Heapsort – Analyse

- Die Worst-Case Komplexität von *Heapify* ist maximal $2 \cdot \lfloor \log_2 n \rfloor$ für n Knoten.
  - für einen Heap mit Level k, gibt es $2 \cdot k$ Vergleiche im Worst-Case
- Die Worst-Case Komplexität von *buildHeap* ist $\Theta(n)$

  (Beweis: spätere Folie)
- Für Heapsort erhalten wir somit:

$$
\begin{aligned}
W(n) &= \left(\sum_{i=1}^{n-1} 2 \cdot \lfloor \log_2 i \rfloor \right) + n \\
&\leqslant 2 \cdot \left(\sum_{i=1}^{n-1} \log_2 n \right) + n \\
&= n + 2 \cdot (n-1) \cdot \log_2 n
\end{aligned}
$$

$$\Rightarrow W(n) \in O(n \cdot \log n)$$

- Zusätzlicher Speicherplatzbedarf ist konstant (lokale Variablen).

Heapsort – Heapeigenschaften

Lemma

Ein n-elementiger Heap hat die Höhe $\lfloor \log_2 n \rfloor$.

Heapsort – Heapeigenschaften

Lemma

Ein n-elementiger Heap hat die Höhe $\lfloor \log_2 n \rfloor$.

Lemma

Ein Heap hat maximal $\lceil n/2^{h+1} \rceil$ *Knoten mit der Höhe h.*

↳ Induktion

Heapsort – Heapeigenschaften

Lemma

Ein n-elementiger Heap hat die Höhe $\lfloor \log_2 n \rfloor$.

Lemma

Ein Heap hat maximal $\lceil n/2^{h+1} \rceil$ *Knoten mit der Höhe* h.

Beweise siehe Übung

Heapsort – Komplexitätsanalyse

Die Worst-Case Komplexität von *buildHeap* ist $\Theta(n)$

Beweis:

Heapsort – Komplexitätsanalyse

$\boxed{\text{Die Worst-Case Komplexität von } \textit{buildHeap} \text{ ist } \Theta(n)}$

log n

Beweis:

- Die Laufzeit von *Heapify* für einen Knoten der Höhe h ist in $O(h)$.

Heapsort – Komplexitätsanalyse

Die Worst-Case Komplexität von *buildHeap* ist $\Theta(n)$

Beweis:

- Die Laufzeit von *Heapify* für einen Knoten der Höhe h ist in $O(h)$.
- $\lceil n/2^{h+1} \rceil$ = Anzahl der Knoten mit Höhe h.
  Daraus folgt für *buildHeap*:

Heapsort – Komplexitätsanalyse

$\boxed{\text{Die Worst-Case Komplexität von } \textit{buildHeap} \text{ ist } \Theta(n)}$

Beweis:

- Die Laufzeit von *Heapify* für einen Knoten der Höhe h ist in $O(h)$.
- $\lceil n/2^{h+1} \rceil$ = Anzahl der Knoten mit Höhe h. (*)
  Daraus folgt für *buildHeap*:

$$\underbrace{\sum_{h=0}^{\lfloor \log_2(n) \rfloor}}_{\text{betrachte alle Ebenen}} \underbrace{\left\lceil \frac{n}{2^{h+1}} \right\rceil}_{(*)} \underbrace{\mathcal{O}(h)}_{\text{Laufzeit Heapify}}$$

Heapsort – Komplexitätsanalyse

$\boxed{\text{Die Worst-Case Komplexität von } \textit{buildHeap} \text{ ist } \Theta(n)}$

Beweis:

- Die Laufzeit von *Heapify* für einen Knoten der Höhe h ist in $O(h)$.
- $\lceil n/2^{h+1} \rceil$ = Anzahl der Knoten mit Höhe h.
  Daraus folgt für *buildHeap*:

$$\underbrace{\sum_{h=0}^{\lfloor \log_2(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathcal{O}(h)} = \mathcal{O}\left(n \sum_{h=0}^{\lfloor \log_2(n) \rfloor} \frac{h}{2^h} \right)$$

$$\mathcal{O}\left(\sum \left\lceil \frac{n}{2^{h+1}} \right\rceil h \right)$$

$$= \mathcal{O}\left(n \cdot \sum \left\lceil \frac{h}{2^{h+1}} \right\rceil \right) = \mathcal{O}\left(n \cdot \sum \frac{h}{2^h} \right)$$

Heapsort – Komplexitätsanalyse

$\boxed{\text{Die Worst-Case Komplexität von } \textit{buildHeap} \text{ ist } \Theta(n)}$

Beweis:

- Die Laufzeit von *Heapify* für einen Knoten der Höhe h ist in $O(h)$.
- $\lceil n/2^{h+1} \rceil$ = Anzahl der Knoten mit Höhe h.
  Daraus folgt für *buildHeap*:

$$\sum_{h=0}^{\lfloor \log_2(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathcal{O}(h) = \mathcal{O}\left(n \sum_{h=0}^{\lfloor \log_2(n) \rfloor} \frac{h}{2^h} \right) \quad \Big| \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

$$\frac{0}{1} + \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \frac{5}{32} + \frac{6}{64} + \ldots\ldots$$

Heapsort – Komplexitätsanalyse

Die Worst-Case Komplexität von *buildHeap* ist $\Theta(n)$

Beweis:

- Die Laufzeit von *Heapify* für einen Knoten der Höhe h ist in $O(h)$.
- $\lceil n/2^{h+1} \rceil$ = Anzahl der Knoten mit Höhe h.
  Daraus folgt für *buildHeap*:

$$\sum_{h=0}^{\lfloor \log_2(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathcal{O}(h) = \mathcal{O}\left(n \sum_{h=0}^{\lfloor \log_2(n) \rfloor} \frac{h}{2^h} \right) \quad | \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

$$= \mathcal{O}\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

Heapsort – Komplexitätsanalyse

Die Worst-Case Komplexität von *buildHeap* ist $\Theta(n)$

Beweis:

- Die Laufzeit von *Heapify* für einen Knoten der Höhe h ist in $O(h)$.
- $\lceil n/2^{h+1} \rceil$ = Anzahl der Knoten mit Höhe h.
  Daraus folgt für *buildHeap*:

$$\sum_{h=0}^{\lfloor \log_2(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathcal{O}(h) = \mathcal{O}\left(n \sum_{h=0}^{\lfloor \log_2(n) \rfloor} \frac{h}{2^h} \right) \quad \Big| \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

$$= \mathcal{O}\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

$$= \mathcal{O}(n)$$

Heapsort – Zusammenfassung

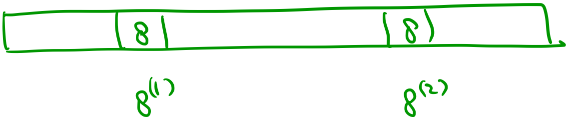
- Heapsort sortiert in $\mathcal{O}(n \cdot \log n)$.

Heapsort – Zusammenfassung

- Heapsort sortiert in $\mathcal{O}(n \cdot \log n)$.
- Heapsort ist ein in-place Algorithmus.

Heapsort – Zusammenfassung

- Heapsort sortiert in $\mathcal{O}(n \cdot \log n)$.
- Heapsort ist ein in-place Algorithmus.
- Heapsort ist nicht stabil.



Übersicht

Erinnerung: Die Prioritätswarteschlange (I)

- Betrachte Elemente, die mit einem Schlüssel (key) versehen sind.

Erinnerung: Die Prioritätswarteschlange (I)

- Betrachte Elemente, die mit einem Schlüssel (key) versehen sind.
- Jeder Schlüssel sei höchstens an ein Element vergeben.

Erinnerung: Die Prioritätswarteschlange (I)

- Betrachte Elemente, die mit einem Schlüssel (key) versehen sind.
- Jeder Schlüssel sei höchstens an ein Element vergeben.
- Schlüssel werden als Priorität betrachtet.

Erinnerung: Die Prioritätswarteschlange (I)

- Betrachte Elemente, die mit einem Schlüssel (key) versehen sind.
- Jeder Schlüssel sei höchstens an ein Element vergeben.
- Schlüssel werden als Priorität betrachtet.
- Die Elemente werden nach ihrer Priorität sortiert.

Erinnerung: Die Prioritätswarteschlange (II)

Prioritätswarteschlange (priority queue)

- `void insert(PriorityQueue pq, Element e, int k)` fügt das Element `e` mit dem Schlüssel `k` in `pq` ein.

Erinnerung: Die Prioritätswarteschlange (II)

Prioritätswarteschlange (priority queue)

- `void insert(PriorityQueue pq, Element e, int k)` fügt das Element `e` mit dem Schlüssel `k` in `pq` ein.
- `Element getMin(PriorityQueue pq)` gibt das Element mit dem kleinsten Schlüssel zurück; benötigt nicht-leere `pq`.

Erinnerung: Die Prioritätswarteschlange (II)

Prioritätswarteschlange (priority queue)

- `void insert(PriorityQueue pq, Element e, int k)` fügt das Element `e` mit dem Schlüssel `k` in `pq` ein.
- `Element getMin(PriorityQueue pq)` gibt das Element mit dem kleinsten Schlüssel zurück; benötigt nicht-leere `pq`.
- `void delMin(PriorityQueue pq)` entfernt das Element mit dem kleinsten Schlüssel; benötigt nicht-leere `pq`.

Erinnerung: Die Prioritätswarteschlange (II)

Prioritätswarteschlange (priority queue)

- `void insert(PriorityQueue pq, Element e, int k)` fügt das Element `e` mit dem Schlüssel `k` in `pq` ein.
- `Element getMin(PriorityQueue pq)` gibt das Element mit dem kleinsten Schlüssel zurück; benötigt nicht-leere `pq`.
- `void delMin(PriorityQueue pq)` entfernt das Element mit dem kleinsten Schlüssel; benötigt nicht-leere `pq`.
- `Element getElt(PriorityQueue pq, int k)` gibt das Element `e` mit dem Schlüssel `k` aus `pq` zurück; `k` muss in `pq` enthalten sein.

Erinnerung: Die Prioritätswarteschlange (II)

Prioritätswarteschlange (priority queue)

- `void insert(PriorityQueue pq, Element e, int k)` fügt das Element `e` mit dem Schlüssel `k` in `pq` ein.
- `Element getMin(PriorityQueue pq)` gibt das Element mit dem kleinsten Schlüssel zurück; benötigt nicht-leere `pq`.
- `void delMin(PriorityQueue pq)` entfernt das Element mit dem kleinsten Schlüssel; benötigt nicht-leere `pq`.
- `Element getElt(PriorityQueue pq, int k)` gibt das Element `e` mit dem Schlüssel `k` aus `pq` zurück; `k` muss in `pq` enthalten sein.
- `void decrKey(PriorityQueue pq, Element e, int k)` setzt den Schlüssel von Element `e` auf `k`; `e` muss in `pq` enthalten sein. `k` muss außerdem kleiner als der bisherige Schlüssel von `e` sein.

Erinnerung: Die Prioritätswarteschlange (II)

Prioritätswarteschlange (priority queue)

- `void insert(PriorityQueue pq, Element e, int k)` fügt das Element `e` mit dem Schlüssel `k` in `pq` ein.
- `Element getMin(PriorityQueue pq)` gibt das Element mit dem kleinsten Schlüssel zurück; benötigt nicht-leere `pq`.
- `void delMin(PriorityQueue pq)` entfernt das Element mit dem kleinsten Schlüssel; benötigt nicht-leere `pq`.
- `Element getElt(PriorityQueue pq, int k)` gibt das Element `e` mit dem Schlüssel `k` aus `pq` zurück; `k` muss in `pq` enthalten sein.
- `void decrKey(PriorityQueue pq, Element e, int k)` setzt den Schlüssel von Element `e` auf `k`; `e` muss in `pq` enthalten sein. `k` muss außerdem kleiner als der bisherige Schlüssel von `e` sein.

Mit Heaps ist eine effiziente Implementierung möglich.

Drei Prioritätswarteschlangenimplementierungen

	Implementierung		
Operation	unsortiertes Array	sortiertes Array	Heap
`isEmpty(pq)`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`insert(pq,e,k)`	$\Theta(1)$	$\Theta(n)^*$	$\Theta(\log(n))$
`getMin(pq)`	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
`delMin(pq)`	$\Theta(n)^*$	$\Theta(1)$	$\Theta(\log(n))$
`getElt(pq,k)`	$\Theta(n)$	$\Theta(\log(n))^\dagger$	$\Theta(n)$
`decrKey(pq,e,k)`	$\Theta(1)$	$\Theta(n)^*$	$\Theta(\log(n))$

*Beinhaltet das Verschieben aller Elemente „rechts“ von `k`.

†Mittels binärer Suche.

Nächste Vorlesung

Nächste Vorlesung

Freitag 18. Mai, 13:15 (Hörsaal H01). Bis dann!