

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
- 3 Heapsort
- 4 Anwendung: Prioritätswarteschlangen

Übersicht

- 1 Heaps
- 2 Heapaufbau
- 3 Heapsort
- 4 Anwendung: Prioritätswarteschlangen

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:

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.

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.

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-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).
- ▶ Die Blätter der untersten Ebene sind linksbündig angeordnet.

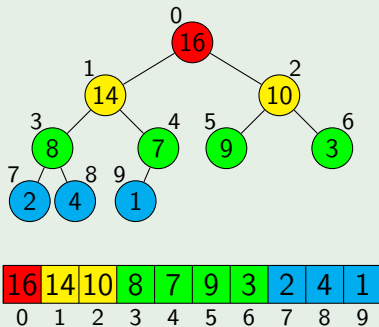
Arrayeinbettung eines Heaps

Arrayeinbettung

Das Array a wird wie folgt als Binärbaum aufgefasst:

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

Beispiel



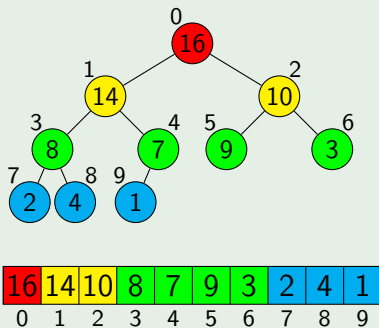
Arrayeinbettung eines Heaps

Arrayeinbettung

Das Array a wird wie folgt als Binärbaum aufgefasst:

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

Beispiel



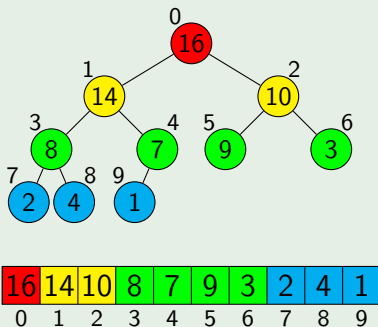
Arrayeinbettung eines Heaps

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



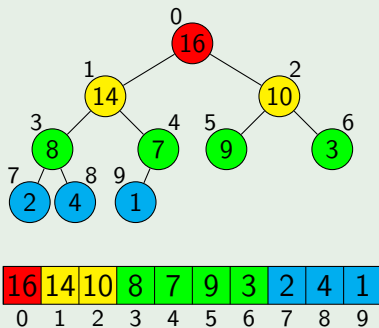
Arrayeinbettung eines Heaps

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.

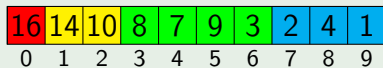
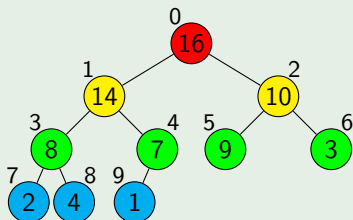
Arrayeinbettung eines Heaps

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

Lemma

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

Lemma

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

Heaps – Eigenschaften

Lemma

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

Lemma

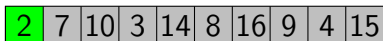
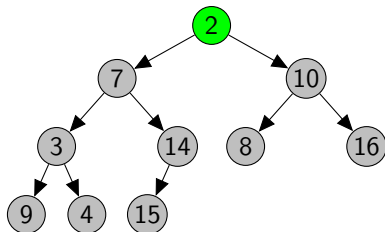
Jedes Array „ist ein Heap“ ab Position $\lfloor \frac{n}{2} \rfloor$.

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

Übersicht

- 1 Heaps
- 2 Heapaufbau**
- 3 Heapsort
- 4 Anwendung: Prioritätswarteschlangen

Naiver Heapaufbau

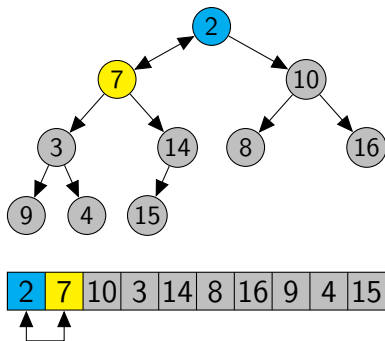


Heapaufbau, naiv

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

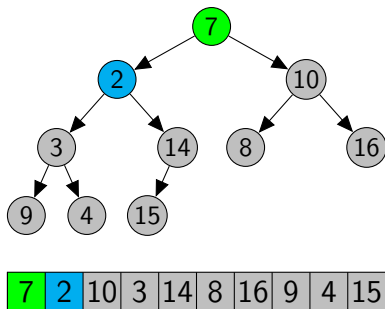


Heapaufbau, naiv

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

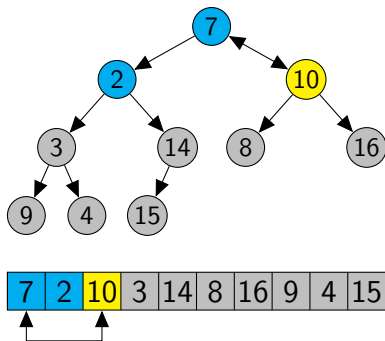


Heapaufbau, naiv

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

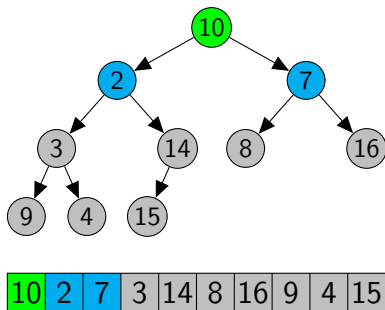


Heapaufbau, naiv

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

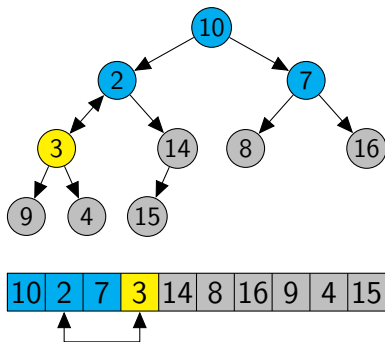


Heapaufbau, naiv

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

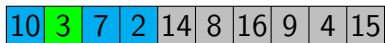
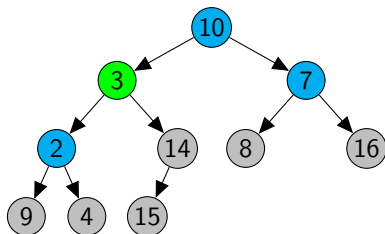


Heapaufbau, naiv

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

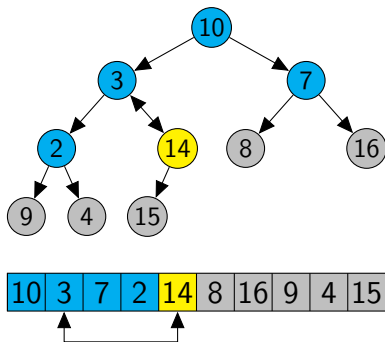


Heapaufbau, naiv

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

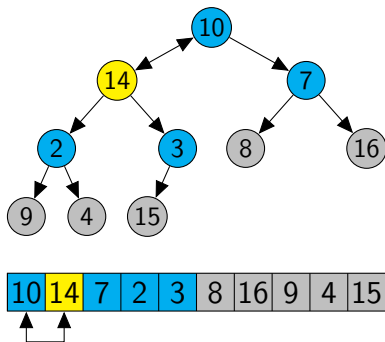


Heapaufbau, naiv

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

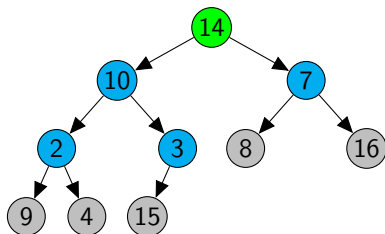


Heapaufbau, naiv

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

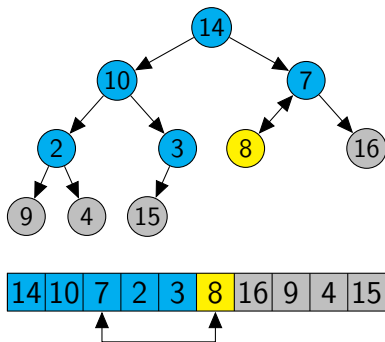


Heapaufbau, naiv

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

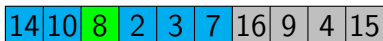
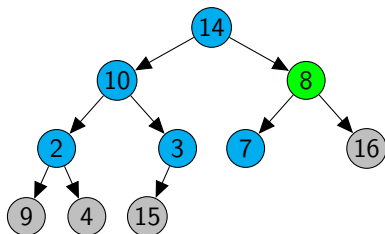


Heapaufbau, naiv

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

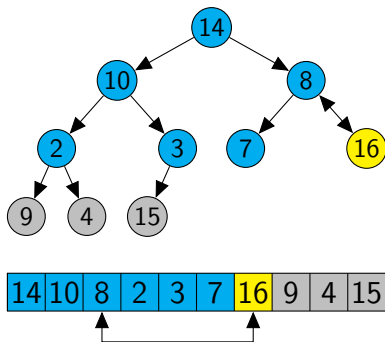


Heapaufbau, naiv

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

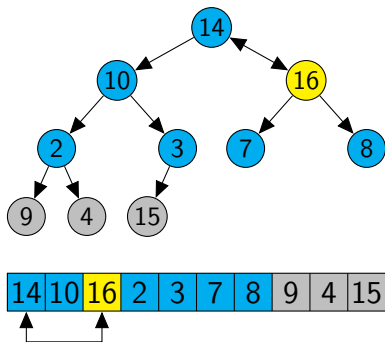


Heapaufbau, naiv

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

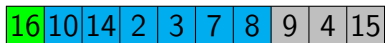
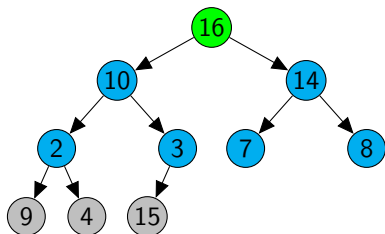


Heapaufbau, naiv

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

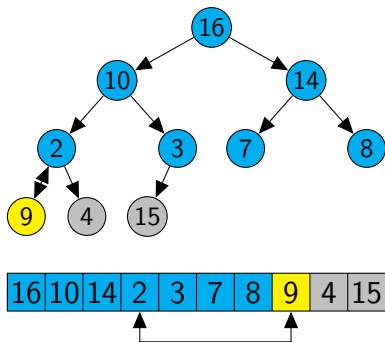


Heapaufbau, naiv

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

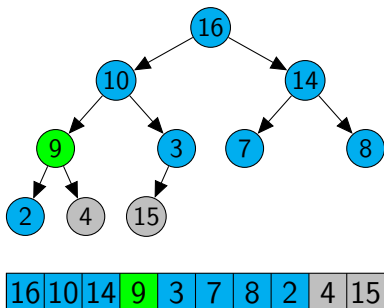


Heapaufbau, naiv

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

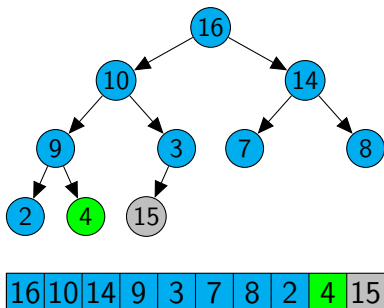


Heapaufbau, naiv

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

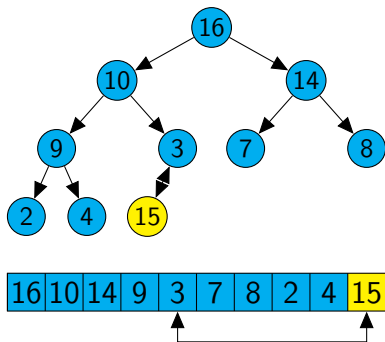


Heapaufbau, naiv

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

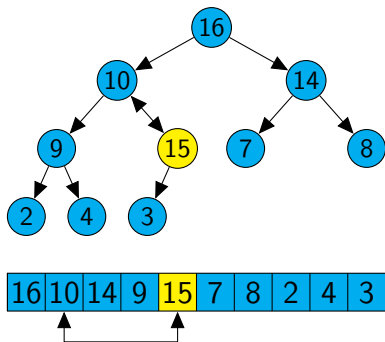


Heapaufbau, naiv

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

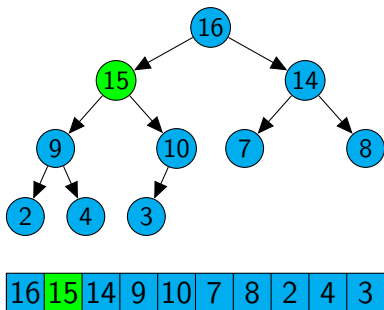


Heapaufbau, naiv

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



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

```
1 void bubble(int E[], int pos) {
2   while (pos > 0) {
3     int parent = (pos - 1) / 2;
4     if (E[parent] > E[pos]) {
5       break;
6     }
7     swap(E[parent], E[pos]);
8     pos = parent;
9   }
10 }
```

Naiver Heapaufbau – Algorithmus und Analyse

```
1 void bubble(int E[], int pos) {
2   while (pos > 0) {
3     int parent = (pos - 1) / 2;
4     if (E[parent] > E[pos]) {
5       break;
6     }
7     swap(E[parent], E[pos]);
8     pos = parent;
9   }
10 }
```

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

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

Naiver Heapaufbau – Algorithmus und Analyse

```
1 void bubble(int E[], int pos) {
2   while (pos > 0) {
3     int parent = (pos - 1) / 2;
4     if (E[parent] > E[pos]) {
5       break;
6     }
7     swap(E[parent], E[pos]);
8     pos = parent;
9   }
10 }
```

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

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

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

Naiver Heapaufbau – Algorithmus und Analyse

```
1 void bubble(int E[], int pos) {
2   while (pos > 0) {
3     int parent = (pos - 1) / 2;
4     if (E[parent] > E[pos]) {
5       break;
6     }
7     swap(E[parent], E[pos]);
8     pos = parent;
9   }
10 }
```

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

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

- ▶ Damit kostet jedes Einfügen $k \approx \log_2 n$ Vergleiche.
- ⇒ Zum Aufbau eines Heaps mit n Elementen benötigt man $\Theta(n \cdot \log(n))$ Vergleiche.

Naiver Heapaufbau – Algorithmus und Analyse

```
1 void bubble(int E[], int pos) {
2   while (pos > 0) {
3     int parent = (pos - 1) / 2;
4     if (E[parent] > E[pos]) {
5       break;
6     }
7     swap(E[parent], E[pos]);
8     pos = parent;
9   }
10 }
```

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

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

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

⇒ 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

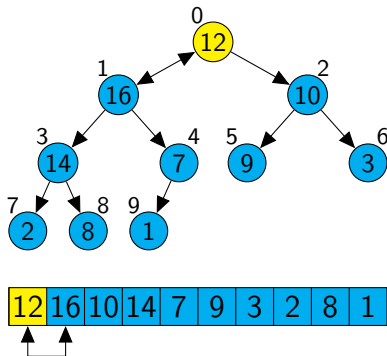
- ▶ Finde das **Maximum** der Werte $E[i]$ und seiner Kinder.
- ▶ Ist $E[i]$ bereits das größte Element, dann ist dieser gesamte Teilbaum auch ein Heap. **Fertig**.
- ▶ Andernfalls **tausche** $E[i]$ mit dem größten Element und führe Heapify in diesem Unterbaum weiter aus.

Heapify – Algorithmus und Beispiel

```

1 void heapify(int E[], int n, int pos) {
2   int next = 2 * pos + 1;
3   while (next < n) {
4     if (next + 1 < n &&
5         E[next + 1] > E[next]) {
6       next = next + 1;
7     }
8     if (E[pos] >= E[next]) {
9       break;
10    }
11    swap(E[pos], E[next]);
12    pos = next;
13    next = 2 * pos + 1;
14  }
15 }

```

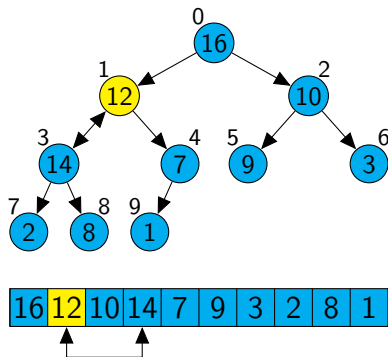


Heapify – Algorithmus und Beispiel

```

1 void heapify(int E[], int n, int pos) {
2   int next = 2 * pos + 1;
3   while (next < n) {
4     if (next + 1 < n &&
5         E[next + 1] > E[next]) {
6       next = next + 1;
7     }
8     if (E[pos] >= E[next]) {
9       break;
10    }
11    swap(E[pos], E[next]);
12    pos = next;
13    next = 2 * pos + 1;
14  }
15 }

```

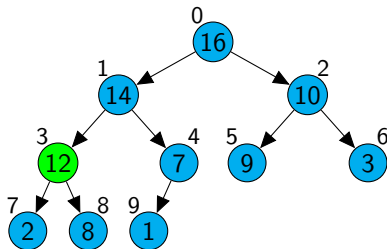


Heapify – Algorithmus und Beispiel

```

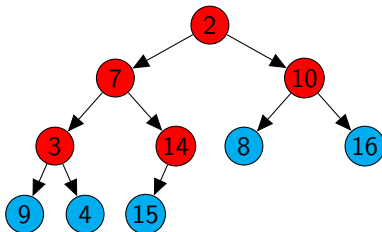
1 void heapify(int E[], int n, int pos) {
2   int next = 2 * pos + 1;
3   while (next < n) {
4     if (next + 1 < n &&
5         E[next + 1] > E[next]) {
6       next = next + 1;
7     }
8     if (E[pos] >= E[next]) {
9       break;
10    }
11    swap(E[pos], E[next]);
12    pos = next;
13    next = 2 * pos + 1;
14  }
15 }

```



Heapaufbau – Algorithmus und Beispiel

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

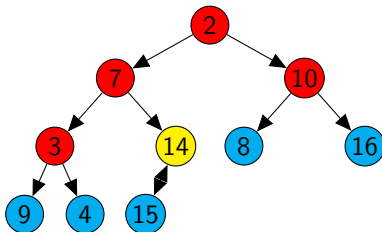


```
1 void buildHeap(int E[]) {  
2   for (int i = E.length / 2 - 1; i >= 0; i--) {  
3     heapify(E, E.length, i);  
4   }  
5 }
```

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

Heapaufbau – Algorithmus und Beispiel

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

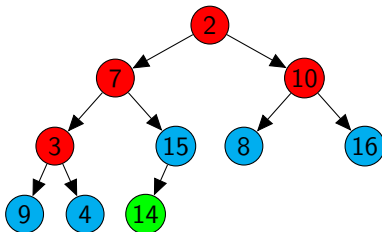


```
1 void buildHeap(int E[]) {  
2   for (int i = E.length / 2 - 1; i >= 0; i--) {  
3     heapify(E, E.length, i);  
4   }  
5 }
```

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

Heapaufbau – Algorithmus und Beispiel

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

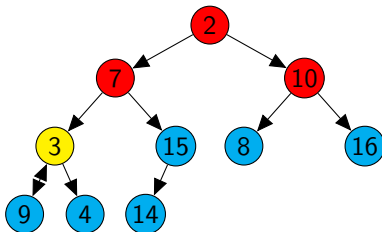


```
1 void buildHeap(int E[]) {
2   for (int i = E.length / 2 - 1; i >= 0; i--) {
3     heapify(E, E.length, i);
4   }
5 }
```

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

Heapaufbau – Algorithmus und Beispiel

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

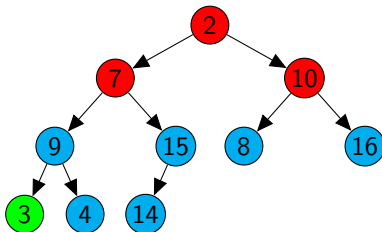


```
1 void buildHeap(int E[]) {  
2   for (int i = E.length / 2 - 1; i >= 0; i--) {  
3     heapify(E, E.length, i);  
4   }  
5 }
```

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

Heapaufbau – Algorithmus und Beispiel

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

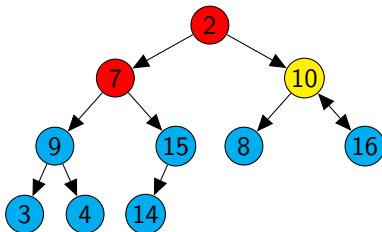


```
1 void buildHeap(int E[]) {
2   for (int i = E.length / 2 - 1; i >= 0; i--) {
3     heapify(E, E.length, i);
4   }
5 }
```

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

Heapaufbau – Algorithmus und Beispiel

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

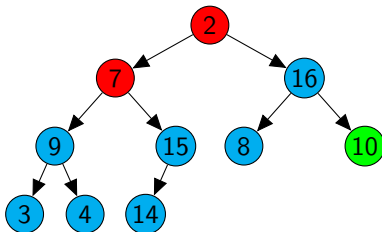


```
1 void buildHeap(int E[]) {
2   for (int i = E.length / 2 - 1; i >= 0; i--) {
3     heapify(E, E.length, i);
4   }
5 }
```

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

Heapaufbau – Algorithmus und Beispiel

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

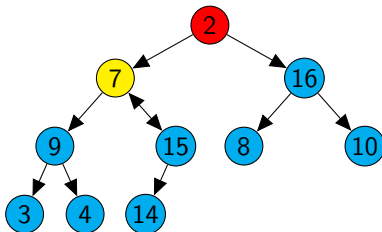


```
1 void buildHeap(int E[]) {
2   for (int i = E.length / 2 - 1; i >= 0; i--) {
3     heapify(E, E.length, i);
4   }
5 }
```

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

Heapaufbau – Algorithmus und Beispiel

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

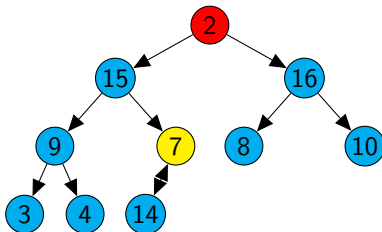


```
1 void buildHeap(int E[]) {
2   for (int i = E.length / 2 - 1; i >= 0; i--) {
3     heapify(E, E.length, i);
4   }
5 }
```

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

Heapaufbau – Algorithmus und Beispiel

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

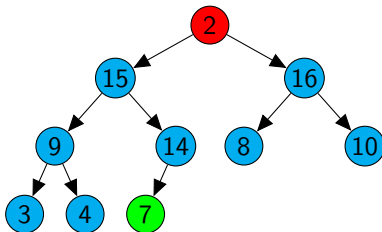


```
1 void buildHeap(int E[]) {
2   for (int i = E.length / 2 - 1; i >= 0; i--) {
3     heapify(E, E.length, i);
4   }
5 }
```

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

Heapaufbau – Algorithmus und Beispiel

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

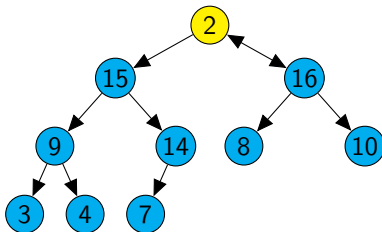


```
1 void buildHeap(int E[]) {
2   for (int i = E.length / 2 - 1; i >= 0; i--) {
3     heapify(E, E.length, i);
4   }
5 }
```

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

Heapaufbau – Algorithmus und Beispiel

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

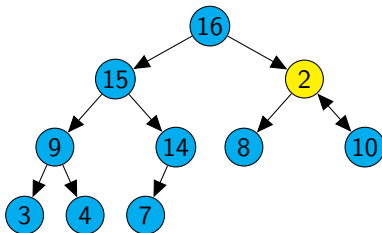


```
1 void buildHeap(int E[]) {
2   for (int i = E.length / 2 - 1; i >= 0; i--) {
3     heapify(E, E.length, i);
4   }
5 }
```

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

Heapaufbau – Algorithmus und Beispiel

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

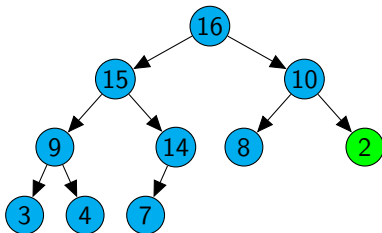


```
1 void buildHeap(int E[]) {  
2   for (int i = E.length / 2 - 1; i >= 0; i--) {  
3     heapify(E, E.length, i);  
4   }  
5 }
```

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

Heapaufbau – Algorithmus und Beispiel

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



```
1 void buildHeap(int E[]) {  
2   for (int i = E.length / 2 - 1; i >= 0; i--) {  
3     heapify(E, E.length, i);  
4   }  
5 }
```

Schleifeninvariant: Nach jedem Aufruf von `heapify(E, E.length, i)` sind die Knoten $i, \dots, E.length - 1$ 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 $i = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \dots$ ist ein Blatt und damit Wurzel eines trivialen Heaps.

Konstruktion eines Heaps

Lemma

Der Algorithmus `buildHeap` ist korrekt und terminiert.

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

Konstruktion eines Heaps

Lemma

Der Algorithmus `buildHeap` ist korrekt und terminiert.

- ▶ Initialisierung: Jeder Knoten $i = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \dots$ ist ein Blatt und damit Wurzel eines trivialen Heaps.
- ▶ Schleifeninvariante: Zu Beginn der `for`-Schleife ist jeder Knoten $i+1, \dots, E.length$ 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 $i = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \dots$ ist ein Blatt und damit Wurzel eines trivialen Heaps.
 - ▶ Schleifeninvariante: Zu Beginn der `for`-Schleife ist jeder Knoten $i+1, \dots, E.length$ die Wurzel eines Heaps.
 - ▶ In jeder Iteration sind alle Kinder des Knotens i bereits Wurzeln eines Heaps (Schleifeninvariante).
- ⇒ Bedingung für den Aufruf von `heapify` ist erfüllt.
- ▶ Dekrementierung von i stellt Schleifeninvariante wieder her.

Konstruktion eines Heaps

Lemma

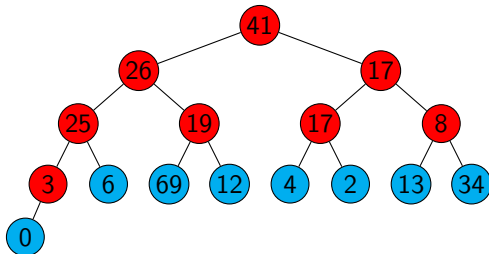
Der Algorithmus `buildHeap` ist korrekt und terminiert.

- ▶ Initialisierung: Jeder Knoten $i = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \dots$ ist ein Blatt und damit Wurzel eines trivialen Heaps.
 - ▶ Schleifeninvariante: Zu Beginn der `for`-Schleife ist jeder Knoten $i+1, \dots, E.length$ die Wurzel eines Heaps.
 - ▶ In jeder Iteration sind alle Kinder des Knotens i bereits Wurzeln eines Heaps (Schleifeninvariante).
- ⇒ Bedingung für den Aufruf von `heapify` ist erfüllt.
- ▶ Dekrementierung von i stellt Schleifeninvariante wieder her.
 - ▶ Terminierung: Bei $i = 0$ ist gemäß Schleifeninvariante jeder Knoten $1, 2, \dots, n$ die Wurzel eines Heaps.

Übersicht

- 1 Heaps
- 2 Heapaufbau
- 3 Heapsort**
- 4 Anwendung: Prioritätswarteschlangen

Heapsort – Algorithmus und Beispiel

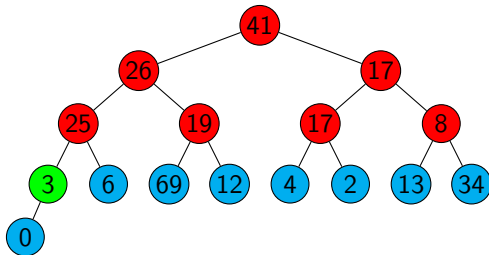


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

Heapsort – Algorithmus und Beispiel

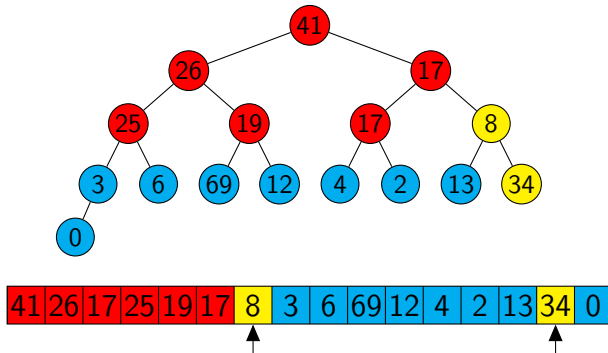


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

Heapsort – Algorithmus und Beispiel

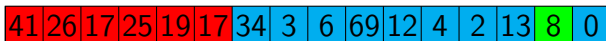
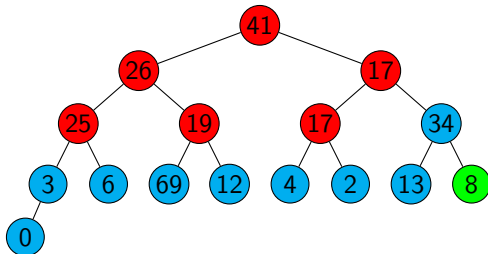


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

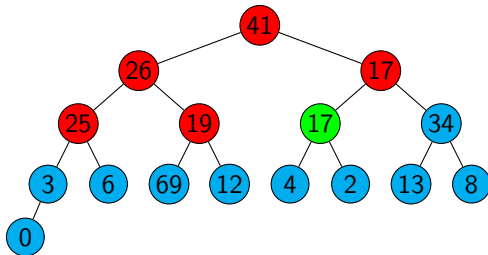
Heapsort – Algorithmus und Beispiel



```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
```

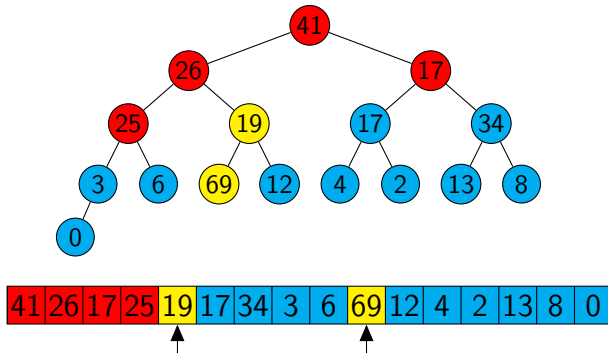
Heapsort – Algorithmus und Beispiel



```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
```

Heapsort – Algorithmus und Beispiel

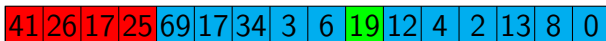
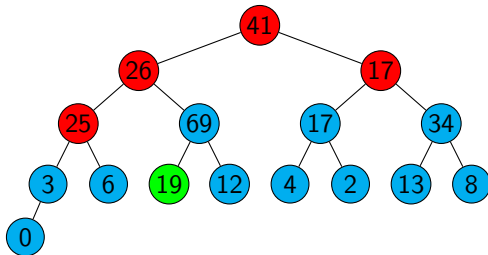


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

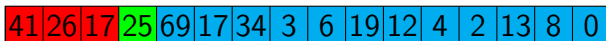
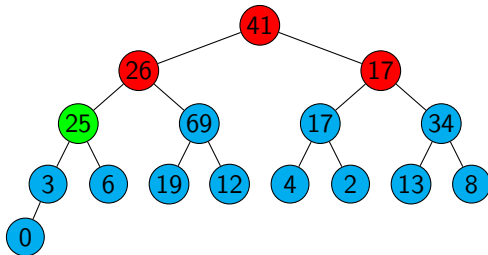
Heapsort – Algorithmus und Beispiel



```

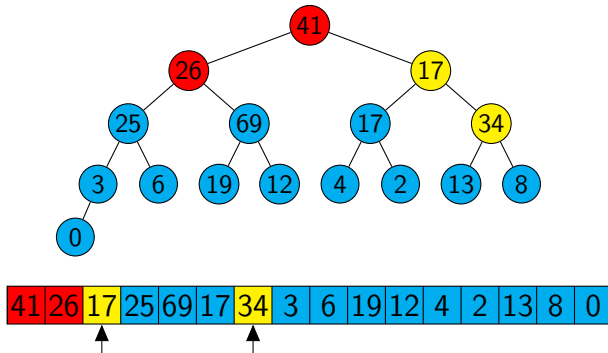
1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
```

Heapsort – Algorithmus und Beispiel



```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```


Heapsort – Algorithmus und Beispiel

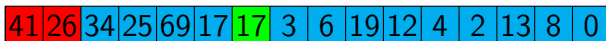
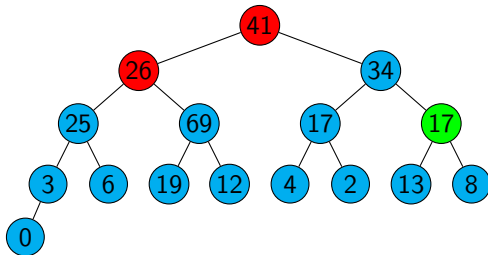


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

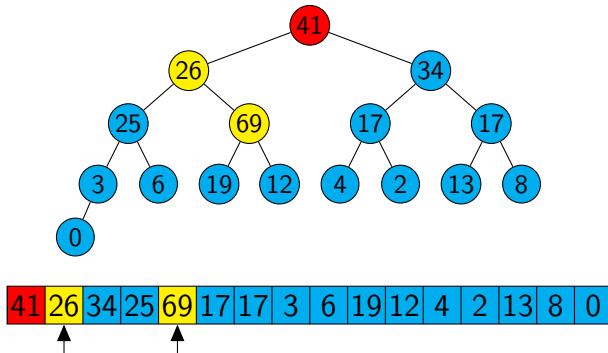
```

Heapsort – Algorithmus und Beispiel



```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel

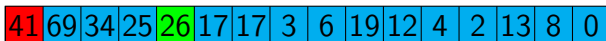
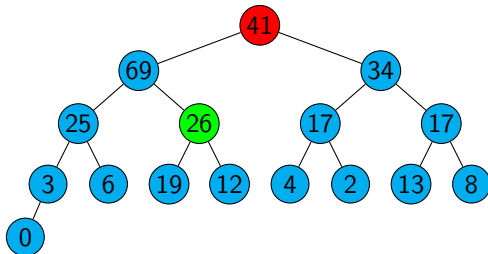


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

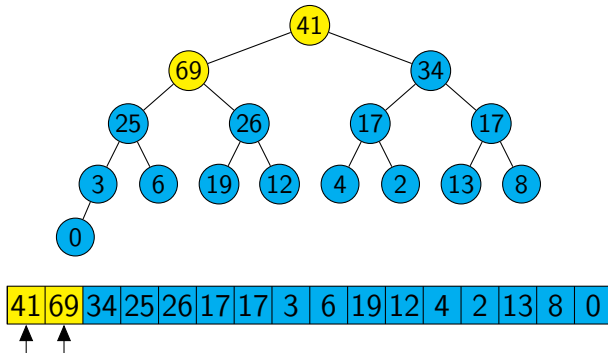
```

Heapsort – Algorithmus und Beispiel



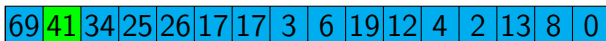
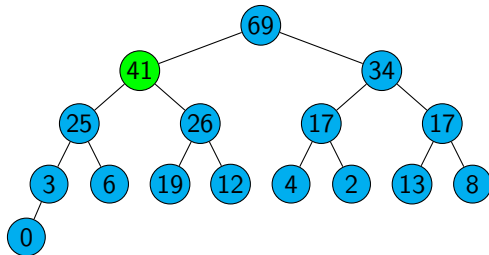
```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel



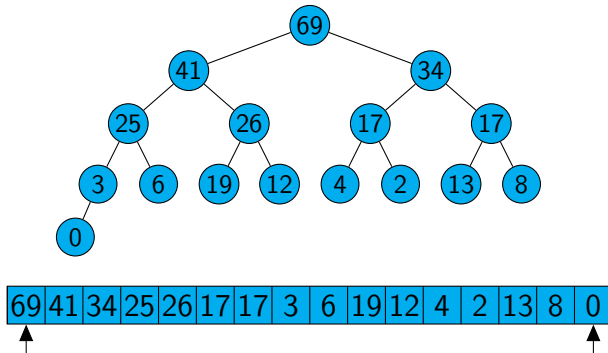
```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel



```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel

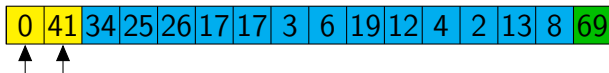
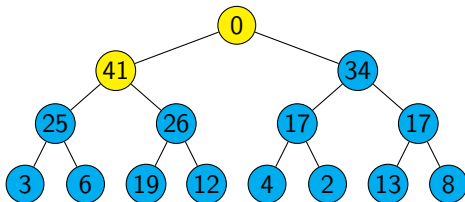


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

Heapsort – Algorithmus und Beispiel

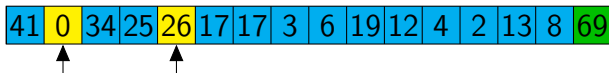
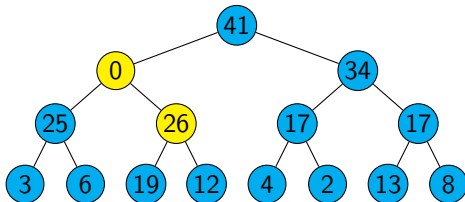


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

Heapsort – Algorithmus und Beispiel

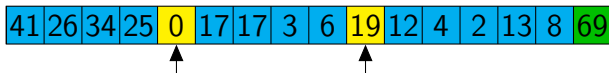
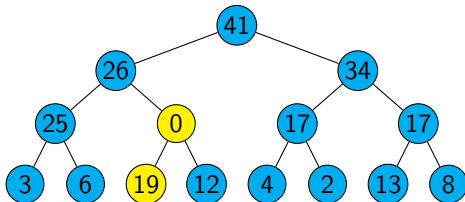


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

Heapsort – Algorithmus und Beispiel

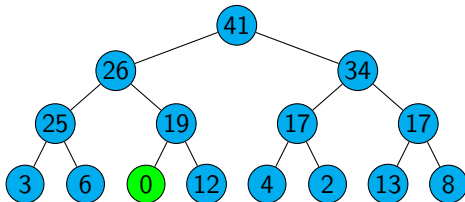


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

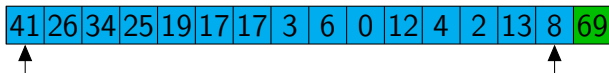
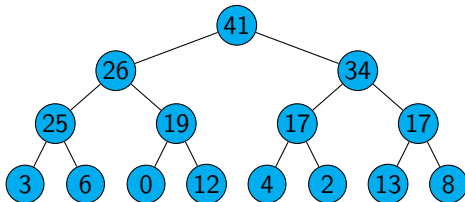
Heapsort – Algorithmus und Beispiel



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

```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel

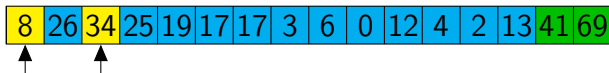
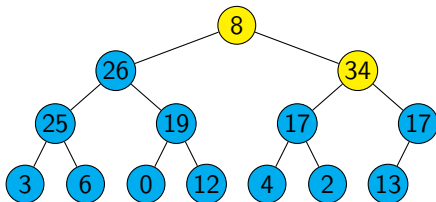


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

Heapsort – Algorithmus und Beispiel

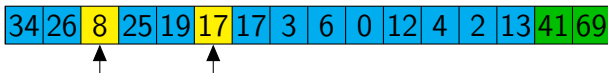
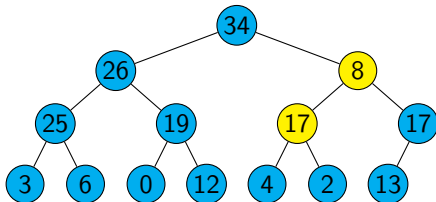


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

Heapsort – Algorithmus und Beispiel

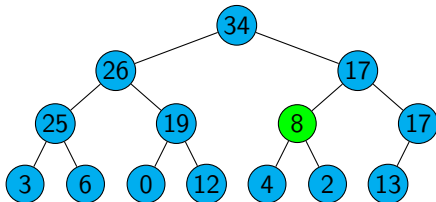


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

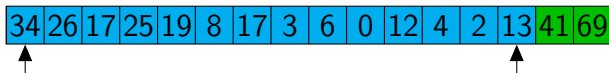
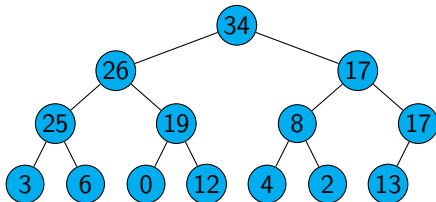
Heapsort – Algorithmus und Beispiel



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

```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel

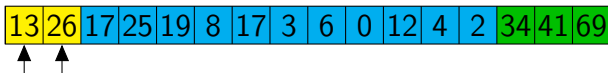
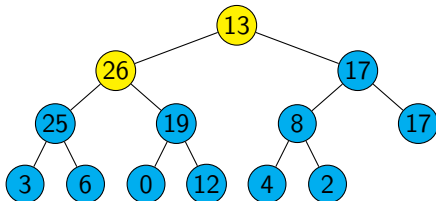


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

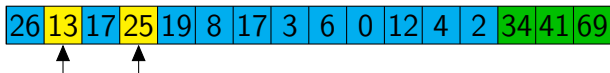
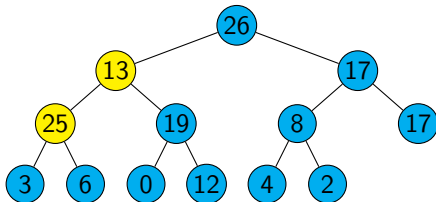

Heapsort – Algorithmus und Beispiel



```

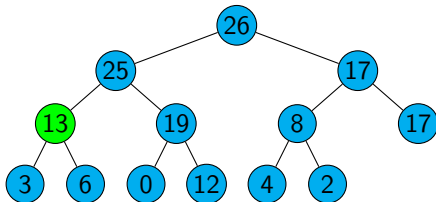
1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
  
```

Heapsort – Algorithmus und Beispiel



```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

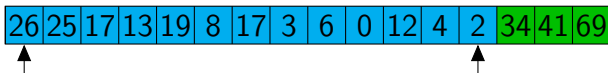
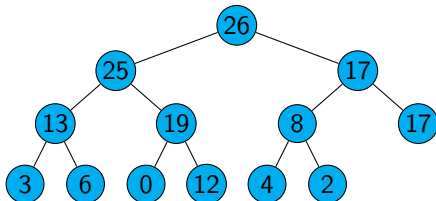
Heapsort – Algorithmus und Beispiel



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

```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel

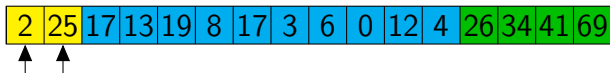
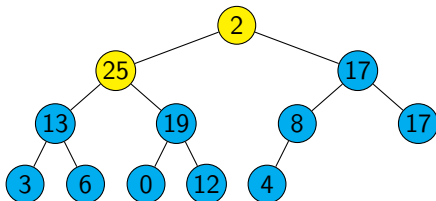


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

Heapsort – Algorithmus und Beispiel

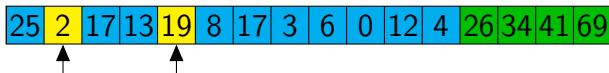
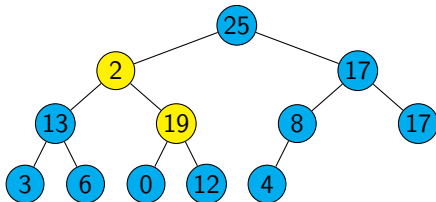


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

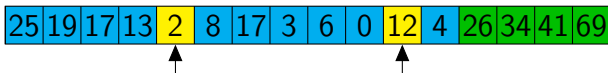
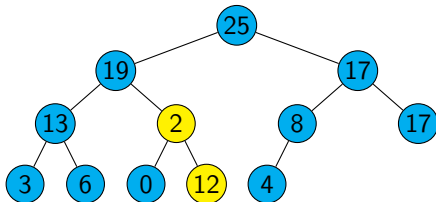
Heapsort – Algorithmus und Beispiel



```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
  
```

Heapsort – Algorithmus und Beispiel

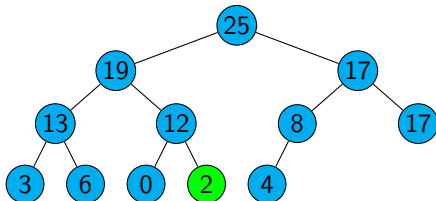


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

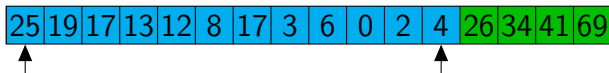
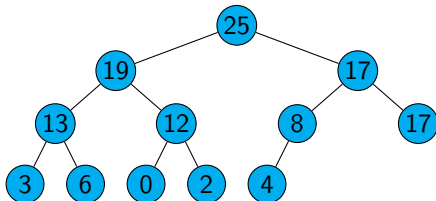
Heapsort – Algorithmus und Beispiel



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

```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```


Heapsort – Algorithmus und Beispiel

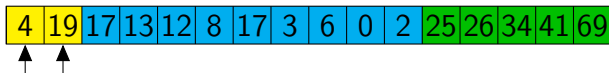
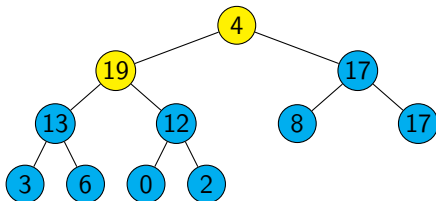


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

Heapsort – Algorithmus und Beispiel

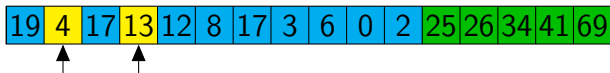
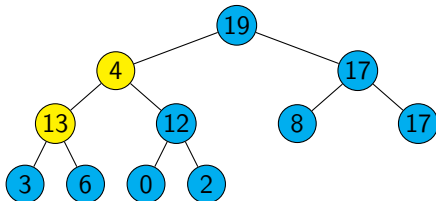


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

Heapsort – Algorithmus und Beispiel

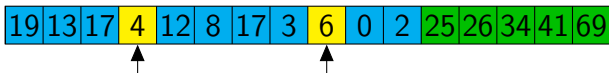
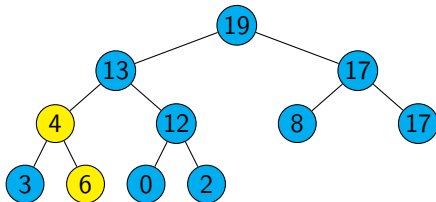


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

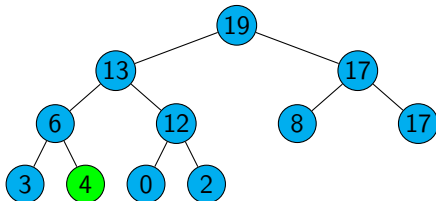
Heapsort – Algorithmus und Beispiel



```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
  
```

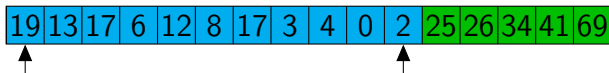
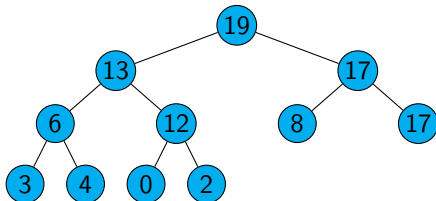
Heapsort – Algorithmus und Beispiel



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

```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

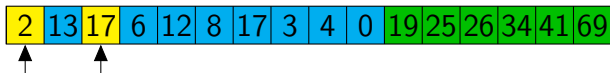
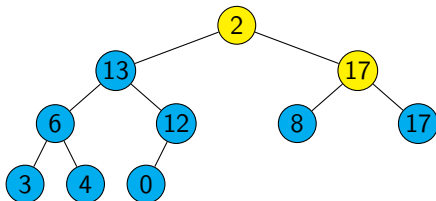
Heapsort – Algorithmus und Beispiel



```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
  
```

Heapsort – Algorithmus und Beispiel

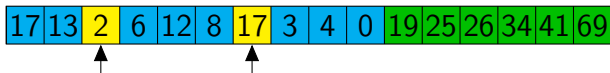
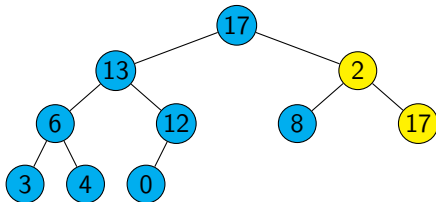


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

Heapsort – Algorithmus und Beispiel

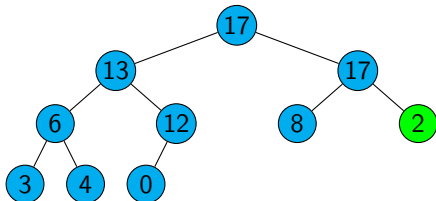


```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }

```

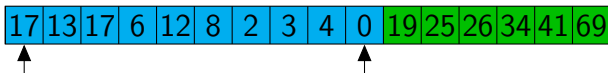
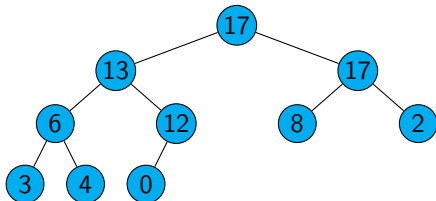
Heapsort – Algorithmus und Beispiel



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

```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

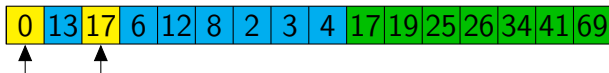
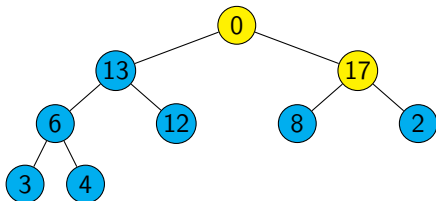
Heapsort – Algorithmus und Beispiel



```

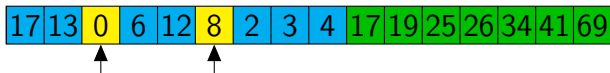
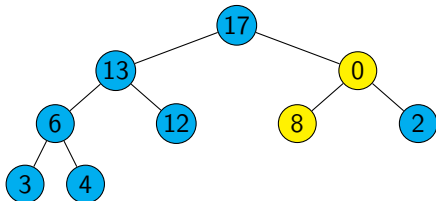
1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
  
```

Heapsort – Algorithmus und Beispiel



```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

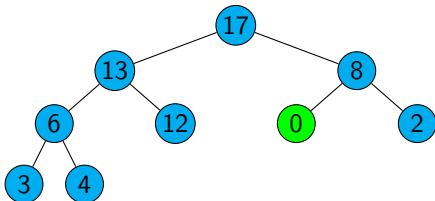
Heapsort – Algorithmus und Beispiel



```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
  
```

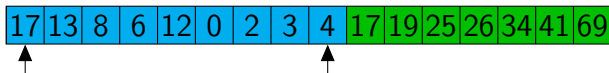
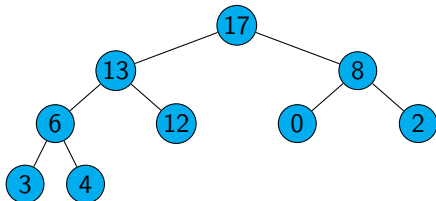
Heapsort – Algorithmus und Beispiel



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

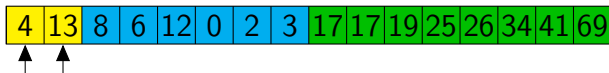
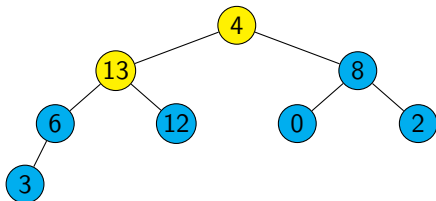
```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel



```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

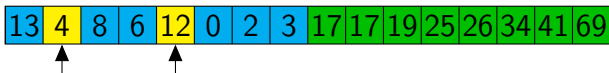
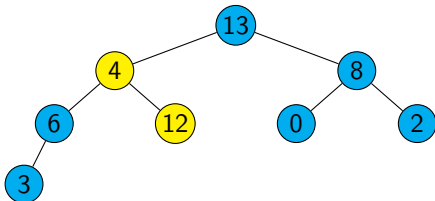
Heapsort – Algorithmus und Beispiel



```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
  
```

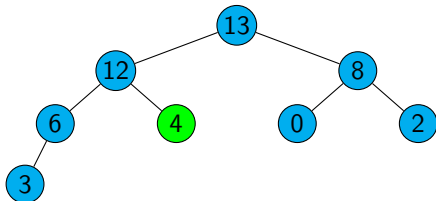
Heapsort – Algorithmus und Beispiel



```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
  
```

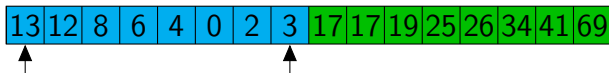
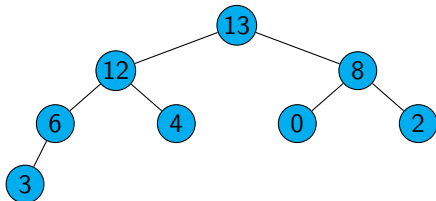
Heapsort – Algorithmus und Beispiel



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

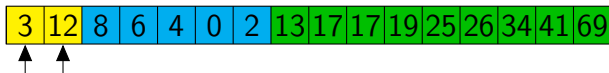
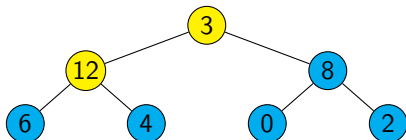
```
1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
```

Heapsort – Algorithmus und Beispiel



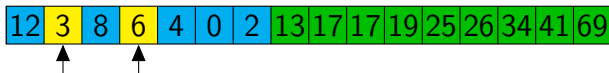
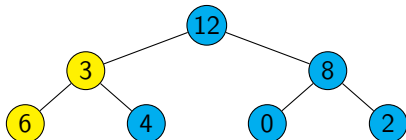
```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel



```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

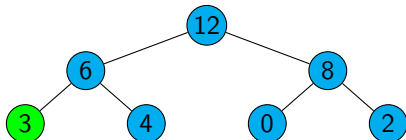
Heapsort – Algorithmus und Beispiel



```

1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
  
```

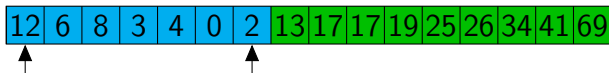
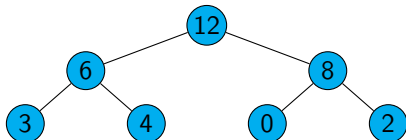
Heapsort – Algorithmus und Beispiel



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

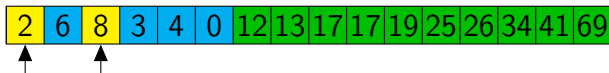
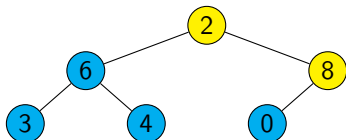
```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel



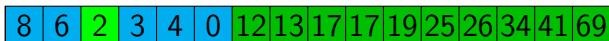
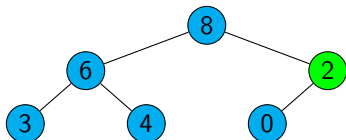
```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel



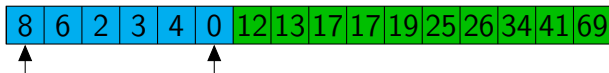
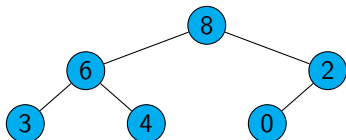
```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel



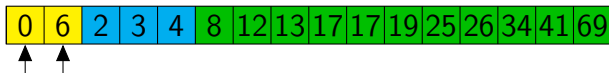
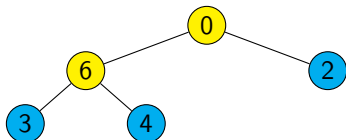
```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```


Heapsort – Algorithmus und Beispiel



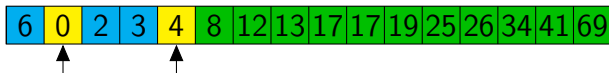
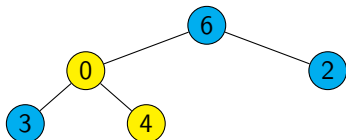
```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel



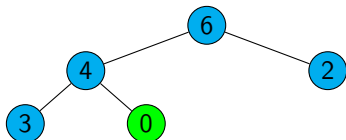
```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel



```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

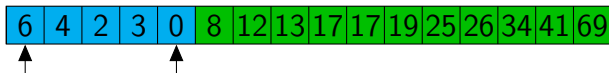
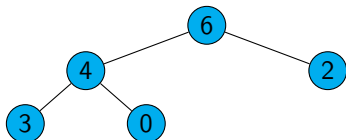
Heapsort – Algorithmus und Beispiel



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

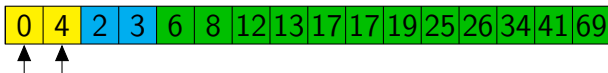
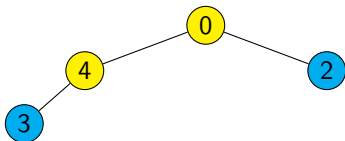
```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel



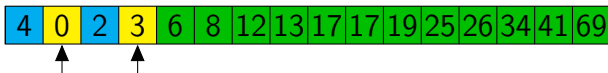
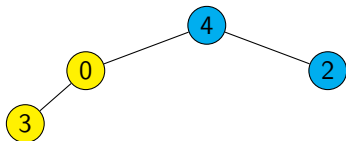
```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel



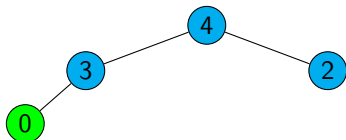
```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel



```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

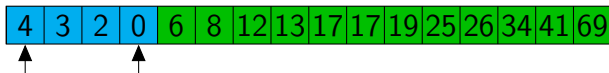
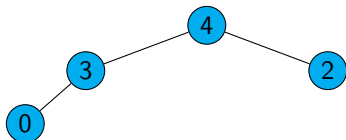
Heapsort – Algorithmus und Beispiel



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

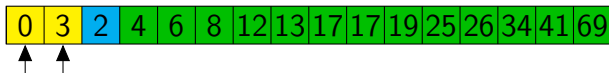
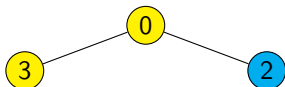
```
1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
```

Heapsort – Algorithmus und Beispiel



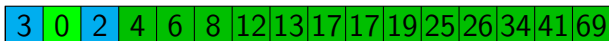
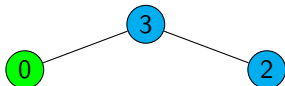
```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel



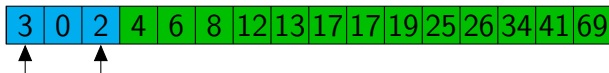
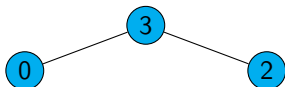
```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel



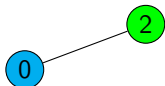
```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel



```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

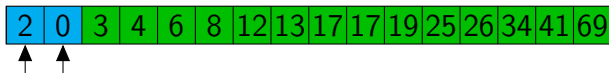
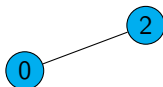
Heapsort – Algorithmus und Beispiel



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

```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel



```
1 void heapSort(int E[]) {  
2   buildHeap(E);  
3   for (int i = E.length - 1; i > 0; i--) {  
4     swap(E[0], E[i]);  
5     heapify(E, i, 0);  
6   }  
7 }
```

Heapsort – Algorithmus und Beispiel

0

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

```
1 void heapSort(int E[]) {
2   buildHeap(E);
3   for (int i = E.length - 1; i > 0; i--) {
4     swap(E[0], E[i]);
5     heapify(E, i, 0);
6   }
7 }
```

Heapsort – Analyse

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

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

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)

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:

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

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:

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

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:

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

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:

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

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

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:

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

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

- ▶ 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 .

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 4.

Heapsort – Komplexitätsanalyse

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

Beweis:

Heapsort – Komplexitätsanalyse

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

Beweis:

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

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

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*:

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

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*:

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

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*:

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

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*:

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

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*:

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

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

- 1 Heaps
- 2 Heapaufbau
- 3 Heapsort
- 4 Anwendung: Prioritätswarteschlangen**

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

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

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

†Mittels binärer Suche.

Nächste Vorlesung

Nächste Vorlesung

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