Max-Heap

Binärer Baum, in dem in jedem Knoten ein Datum gespeichert ist.

Für jeden Knoten v mit Inhalt x gilt, dass alle Daten im **linken oder rechten Teilbaum** von v Schlüssel besitzen, die **höchstens so groß** wie der Schlüssel von x sind.

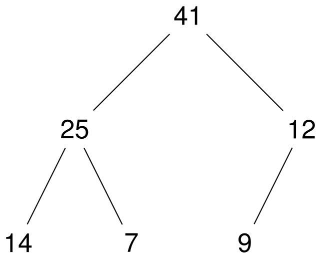
Darüber hinaus ist der Binärbaum bis auf möglicherweise die letzte Ebene **vollständig**. Die letzte Ebene wird **von links nach rechts** gefüllt.

Max-Heap

Binärer Baum, in dem in jedem Knoten ein Datum gespeichert ist.

Für jeden Knoten *v* mit Inhalt *x* gilt, dass alle Daten im **linken oder rechten Teilbaum** von *v* Schlüssel besitzen, die **höchstens so groß** wie der Schlüssel von *x* sind.

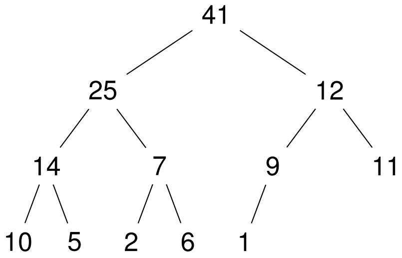
Darüber hinaus ist der Binärbaum bis auf möglicherweise die letzte Ebene **vollständig**. Die letzte Ebene wird **von links nach rechts** gefüllt.



Speichern eines Max-Heaps in einem Feld

Betrachte Max-Heap mit heapsize vielen Elementen.

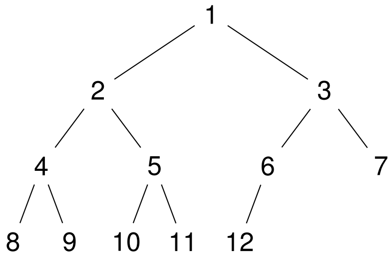
Speichere diesen in Feld a mit Positionen $1, \ldots, \text{heapsize}$.



Speichern eines Max-Heaps in einem Feld

Betrachte Max-Heap mit heapsize vielen Elementen.

Speichere diesen in Feld a mit Positionen $1, \ldots, \text{heapsize}$.

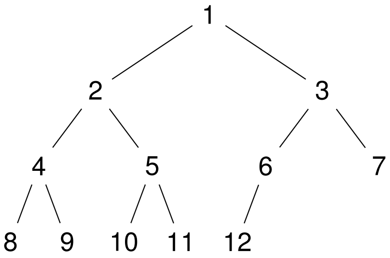


$$a = [41, 25, 12, 14, 7, 9, 11, 10, 5, 2, 6, 1]$$

Speichern eines Max-Heaps in einem Feld

Betrachte Max-Heap mit heapsize vielen Elementen.

Speichere diesen in Feld a mit Positionen $1, \ldots,$ heapsize.



$$a = [41, 25, 12, 14, 7, 9, 11, 10, 5, 2, 6, 1]$$

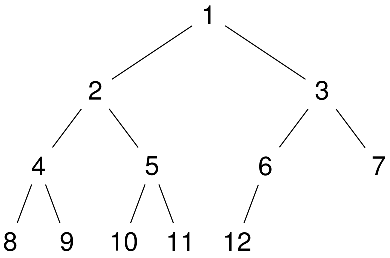
Kinder des Knotens an Position i an Positionen $2i$ und $2i + 1$.

Elternknoten zu dem an Position $i > 1$ gespeicherten Knoten an Stelle $\lfloor i/2 \rfloor$.

Speichern eines Max-Heaps in einem Feld

Betrachte Max-Heap mit heapsize vielen Elementen.

Speichere diesen in Feld a mit Positionen $1, \ldots, \text{heapsize}$.



$$a = [41, 25, 12, 14, 7, 9, 11, 10, 5, 2, 6, 1]$$

Kinder des Knotens an Position i an Positionen $2i$ und $2i + 1$.

Elternknoten zu dem an Position $i > 1$ gespeicherten Knoten an Stelle $\lfloor i/2 \rfloor$.

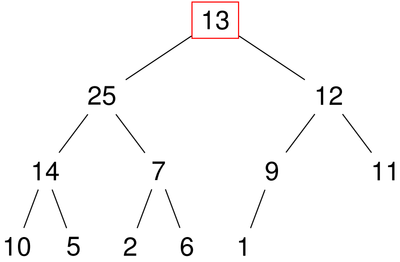
Heap-Bedingung: Für alle i $a[i].\text{key} \geq a[2i].\text{key}$ und $a[i].\text{key} \geq a[2i+1].\text{key}$.

4.4.1 Heap-Eigenschaft herstellen

Erzeugen eines Max-Heaps

Annahme: Heap-Eigenschaft überall erfüllt bis auf evtl. die Wurzel.

```
MAX-HEAPIFY(int i)
    ℓ = 2i; r = 2i + 1; largest = i;
    if (ℓ ≤ heapsize && a[ℓ].key > a[largest].key)
        largest = ℓ;
    if (r ≤ heapsize && a[r].key > a[largest].key)
        largest = r;
    if (largest ≠ i) {
        vertausche a[i] und a[largest];
        MAX-HEAPIFY(largest);
    }
```

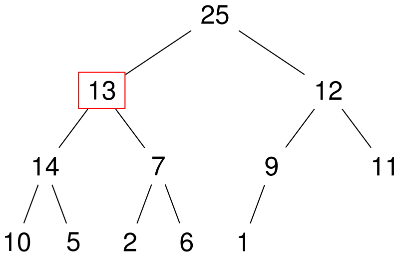


4.4.1 Heap-Eigenschaft herstellen

Erzeugen eines Max-Heaps

Annahme: Heap-Eigenschaft überall erfüllt bis auf evtl. die Wurzel.

```
MAX-HEAPIFY(int i)
    ℓ = 2i; r = 2i + 1; largest = i;
    if (ℓ ≤ heapsize && a[ℓ].key > a[largest].key)
        largest = ℓ;
    if (r ≤ heapsize && a[r].key > a[largest].key)
        largest = r;
    if (largest ≠ i) {
        vertausche a[i] und a[largest];
        MAX-HEAPIFY(largest);
    }
```



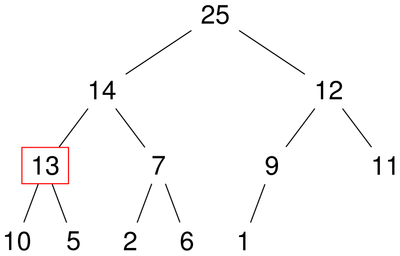
4.4.1 Heap-Eigenschaft herstellen

Erzeugen eines Max-Heaps

Annahme: Heap-Eigenschaft überall erfüllt bis auf evtl. die Wurzel.

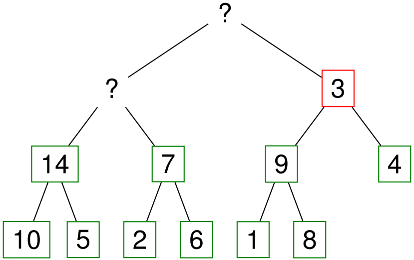
MAX-HEAPIFY(**int** i)

$\ell = 2i$; $r = 2i + 1$; $\text{largest} = i$;
if ($\ell \leq$ heapsize && $a[\ell]$.key $> a[\text{largest}]$.key)
    $\text{largest} = \ell$;
if ($r \leq$ heapsize && $a[r]$.key $> a[\text{largest}]$.key)
    $\text{largest} = r$;
if ($\text{largest} \neq i$) $\{$
    vertausche $a[i]$ und $a[\text{largest}]$;
    MAX-HEAPIFY(largest);
$\}$



Lemma 4.15

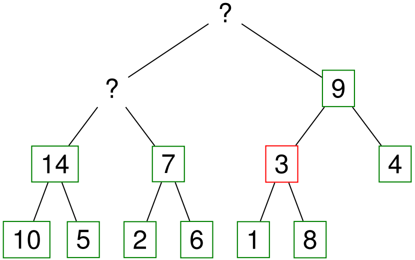
Gilt vor dem Aufruf von MAX-HEAPIFY(i) die Heap-Bedingung für alle $j > i$, so gilt sie anschließend für alle $j \geq i$.



4.4.1 Heap-Eigenschaft herstellen

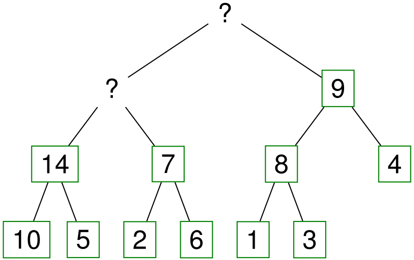
Lemma 4.15

Gilt vor dem Aufruf von MAX-HEAPIFY(i) die Heap-Bedingung für alle $j > i$, so gilt sie anschließend für alle $j \geq i$.



Lemma 4.15

Gilt vor dem Aufruf von Max-Heapify(i) die Heap-Bedingung für alle $j > i$, so gilt sie anschließend für alle $j \geq i$.



4.4.1 Heap-Eigenschaft herstellen

Erzeugen eines Max-Heaps:

```
BUILD-HEAP()
    for (i = ⌊a.length/2⌋; i ≥ 1; i−−) {
        MAX-HEAPIFY(i);
    }
```

4.4.1 Heap-Eigenschaft herstellen

Erzeugen eines Max-Heaps:

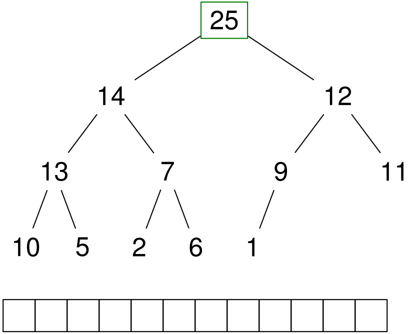
```
BUILD-HEAP()
    for (i = ⌊a.length/2⌋; i ≥ 1; i--) {
        MAX-HEAPIFY(i);
    }
```

Lemma 4.16

Nach der Ausführung von BUILD-HEAP gilt die Heap-Eigenschaft für alle i.

Heapsort

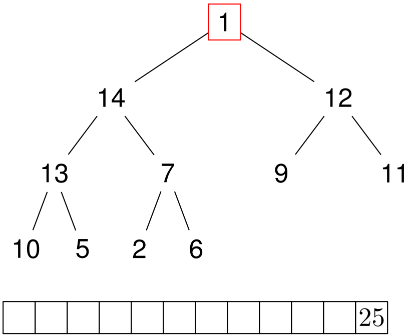
```
HEAPSORT(int[ ] a)
    // Das Feld a besitzt die Positionen 1, . . . , n.
    BUILD-HEAP();
    heapsize = n;
    while (heapsize > 0) {
        vertausche a[1] und a[heapsize];
        heapsize−−;
        MAX-HEAPIFY(1);
    }
```



4.4.2 Heapsort

Heapsort

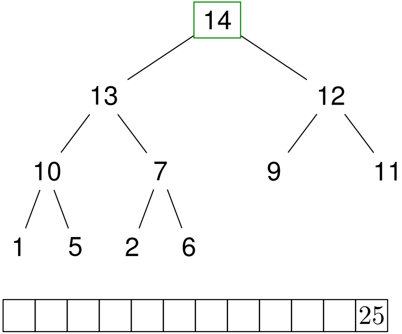
```
HEAPSORT(int[ ] a)
    // Das Feld a besitzt die Positionen 1, . . . , n.
    BUILD-HEAP();
    heapsize = n;
    while (heapsize > 0) {
        vertausche a[1] und a[heapsize];
        heapsize−−;
        MAX-HEAPIFY(1);
    }
```



4.4.2 Heapsort

Heapsort

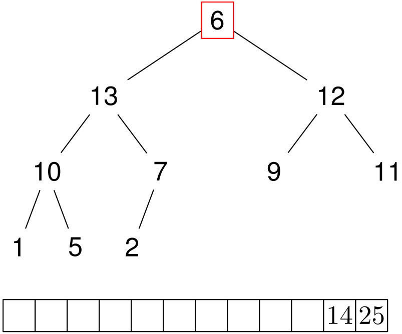
```
HEAPSORT(int[ ] a)
    // Das Feld a besitzt die Positionen 1, . . . , n.
    BUILD-HEAP();
    heapsize = n;
    while (heapsize > 0) {
        vertausche a[1] und a[heapsize];
        heapsize−−;
        MAX-HEAPIFY(1);
    }
```



4.4.2 Heapsort

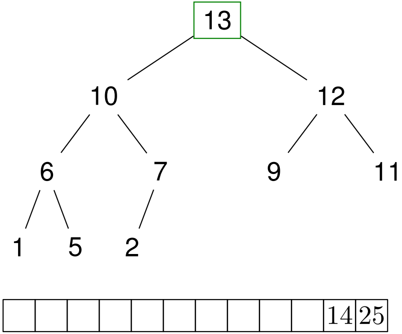
Heapsort

```
HEAPSORT(int[ ] a)
    // Das Feld a besitzt die Positionen 1, ..., n.
    BUILD-HEAP();
    heapsize = n;
    while (heapsize > 0) {
        vertausche a[1] und a[heapsize];
        heapsize−−;
        MAX-HEAPIFY(1);
    }
```



Heapsort

```
HEAPSORT(int[ ] a)
    // Das Feld a besitzt die Positionen 1, . . . , n.
    BUILD-HEAP();
    heapsize = n;
    while (heapsize > 0) {
        vertausche a[1] und a[heapsize];
        heapsize−−;
        MAX-HEAPIFY(1);
    }
```



4.4.2 Heapsort

Heapsort

```
HEAPSORT(int[ ] a)
    // Das Feld a besitzt die Positionen 1, . . . , n.
    BUILD-HEAP();
    heapsize = n;
    while (heapsize > 0) {
        vertausche a[1] und a[heapsize];
        heapsize−−;
        MAX-HEAPIFY(1);
    }
```

1	2	5	6	7	9	10	11	12	13	14	25

4.4.2 Heapsort

Heapsort

```
HEAPSORT(int[ ] a)
    // Das Feld a besitzt die Positionen 1, . . . , n.
    BUILD-HEAP();
    heapsize = n;
    while (heapsize > 0) {
        vertausche a[1] und a[heapsize];
        heapsize−−;
        MAX-HEAPIFY(1);
    }
```

1	2	5	6	7	9	10	11	12	13	14	25

Theorem 4.17

HEAPSORT sortiert jedes Feld mit n Einträgen in Zeit $O(n \log n)$.

4.4.3 Prioritätswarteschlangen

Prioritätswarteschlange

Eine Prioritätswarteschlange speichert **Element/Schlüssel-Paare** und unterstützt die folgenden Operationen.

4.4.3 Prioritätswarteschlangen

Prioritätswarteschlange

Eine Prioritätswarteschlange speichert **Element/Schlüssel-Paare** und unterstützt die folgenden Operationen.

- INSERT(x, k): Füge ein neues Element x mit Schlüssel k ein.

Prioritätswarteschlange

Eine Prioritätswarteschlange speichert **Element/Schlüssel-Paare** und unterstützt die folgenden Operationen.

- $\textsc{Insert}(x, k)$: Füge ein neues Element x mit Schlüssel k ein.
- $\textsc{Find-Max}()$: Gib ein Element mit dem größtem Schlüssel zurück.

Prioritätswarteschlange

Eine Prioritätswarteschlange speichert **Element/Schlüssel-Paare** und unterstützt die folgenden Operationen.

- $\textsc{Insert}(x, k)$: Füge ein neues Element x mit Schlüssel k ein.
- $\textsc{Find-Max}()$: Gib ein Element mit dem größtem Schlüssel zurück.
- $\textsc{Extract-Max}()$: Entferne ein Element mit dem größtem Schlüssel und gib dieses Element zurück.

4.4.3 Prioritätswarteschlangen

Prioritätswarteschlange

Eine Prioritätswarteschlange speichert **Element/Schlüssel-Paare** und unterstützt die folgenden Operationen.

- $\textsc{Insert}(x, k)$: Füge ein neues Element x mit Schlüssel k ein.
- $\textsc{Find-Max}()$: Gib ein Element mit dem größtem Schlüssel zurück.
- $\textsc{Extract-Max}()$: Entferne ein Element mit dem größtem Schlüssel und gib dieses Element zurück.
- $\textsc{Increase-Key}(i, k)$: Erhöhe den Schlüssel des an Stelle i gespeicherten Objektes auf k.

4.4.3 Prioritätswarteschlangen

```
EXTRACT-MAX()
    if (heapsize > 0) {
        max = a[1];
        a[1] = a[heapsize];
        heapsize−−;
        MAX-HEAPIFY(1);
        return max;
    }
```

4.4.3 Prioritätswarteschlangen

```
EXTRACT-MAX()
    if (heapsize > 0) {
        max = a[1];
        a[1] = a[heapsize];
        heapsize−−;
        MAX-HEAPIFY(1);
        return max;
    }
```

```
INCREASE-KEY(i,k)
    a[i].key = k;
    while (i > 1 && a[i].key > a[⌊i/2⌋].key) {
        vertausche a[i] und a[⌊i/2⌋];
        i = ⌊i/2⌋;
    }
```

4.4.3 Prioritätswarteschlangen

```
EXTRACT-MAX()
    if (heapsize > 0) {
        max = a[1];
        a[1] = a[heapsize];
        heapsize−−;
        MAX-HEAPIFY(1);
        return max;
    }
```

```
INCREASE-KEY(i,k)
    a[i].key = k;
    while (i > 1 && a[i].key > a[⌊i/2⌋].key) {
        vertausche a[i] und a[⌊i/2⌋];
        i = ⌊i/2⌋;
    }
```

```
INSERT(x,k)
    heapsize++;
    a[heapsize] = x;
    a[heapsize].key = −∞;
    INCREASE-KEY(heapsize, k);
```

Theorem 4.18

Die Laufzeit der Operationen EXTRACT-MAX, INCREASE-KEY und INSERT beträgt in einem Heap mit n gespeicherten Daten $O(\log n)$. Die Operation FIND-MAX benötigt nur konstante Laufzeit.