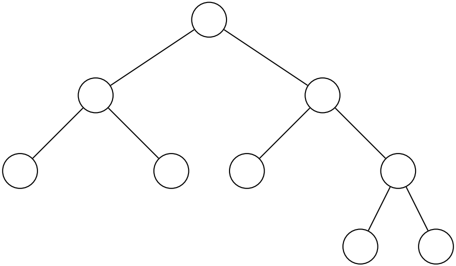
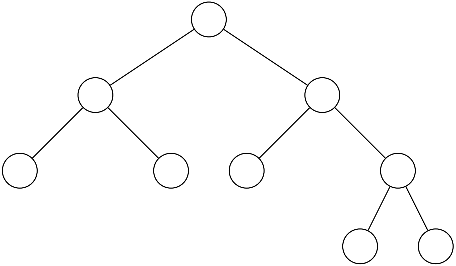
Informatik

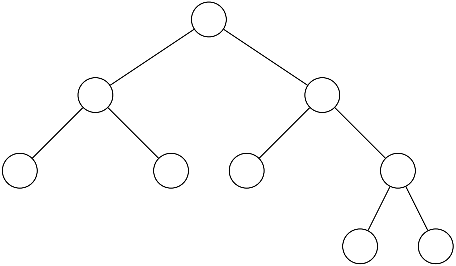
HeapSort



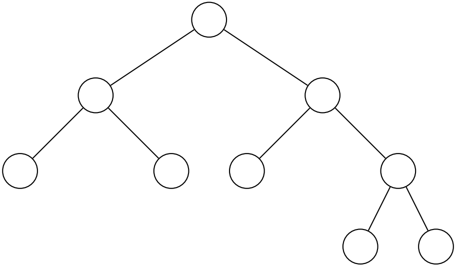
Ein *binärer Baum* ist entweder leer oder besteht aus einem Knoten, dem zwei binäre Bäume zugeordnet sind.



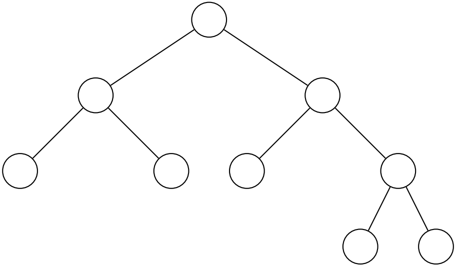
Ein *binärer Baum* ist entweder leer oder besteht aus einem Knoten, dem zwei binäre Bäume zugeordnet sind. Dieser heißt dann der *Elternknoten* des linken bzw. rechten Teilbaums.



Ein *binärer Baum* ist entweder leer oder besteht aus einem Knoten, dem zwei binäre Bäume zugeordnet sind. Dieser heißt dann der *Elternknoten* des linken bzw. rechten Teilbaums. Ein Knoten ohne einen Elternknoten heißt *Wurzel*.



Ein *binärer Baum* ist entweder leer oder besteht aus einem Knoten, dem zwei binäre Bäume zugeordnet sind. Dieser heißt dann der *Elternknoten* des linken bzw. rechten Teilbaums. Ein Knoten ohne einen Elternknoten heißt *Wurzel*. Die Knoten, die x als Elternknoten haben, sind seine *Kinder*. Knoten ohne Kinder heißen *Blätter*.



Ein *binärer Baum* ist entweder leer oder besteht aus einem Knoten, dem zwei binäre Bäume zugeordnet sind. Dieser heißt dann der *Elternknoten* des linken bzw. rechten Teilbaums. Ein Knoten ohne einen Elternknoten heißt *Wurzel*. Die Knoten, die x als Elternknoten haben, sind seine *Kinder*. Knoten ohne Kinder heißen *Blätter*.

Im Beispiel hat der Baum 4 Ebenen. Die Wurzel ist auf Ebene 0, die zwei rechten Blätter sind auf Ebene 3.

Ein *Heap* ist ein binärer Baum für den gilt:

- Jeder Knoten enthält einen Schlüssel

Ein *Heap* ist ein binärer Baum für den gilt:

- Jeder Knoten enthält einen Schlüssel
- Schlüssel im Elternknoten $\leq$ Schlüssel im Kind

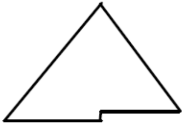
Ein *Heap* ist ein binärer Baum für den gilt:

- Jeder Knoten enthält einen Schlüssel
- Schlüssel im Elternknoten $\leq$ Schlüssel im Kind
- Alle Ebenen sind vollständig gefüllt, bis auf die letzte, die muss von links beginnend gefüllt sein.

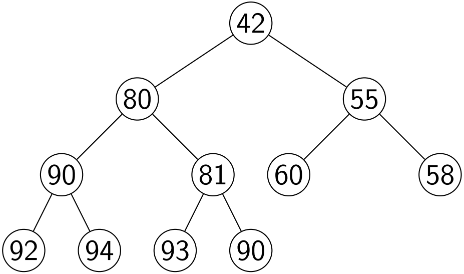
Ein *Heap* ist ein binärer Baum für den gilt:

- Jeder Knoten enthält einen Schlüssel
- Schlüssel im Elternknoten $\leq$ Schlüssel im Kind
- Alle Ebenen sind vollständig gefüllt, bis auf die letzte, die muss von links beginnend gefüllt sein.

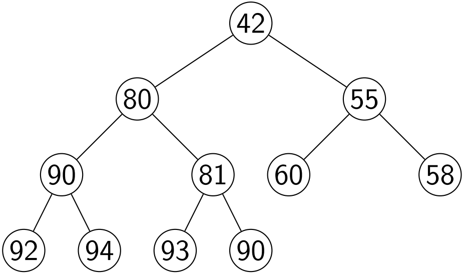
Die Umrisse eines Heaps:



Ein Heap



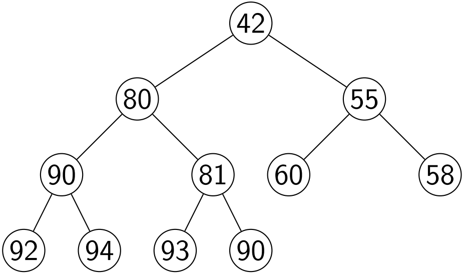
Ein Heap



	42	80	55	90	81	60	58	92	94	93	90
Index:	0	1	2	3	4	5	6	7	8	9	10

linkes Kind von Knoten i:

Ein Heap

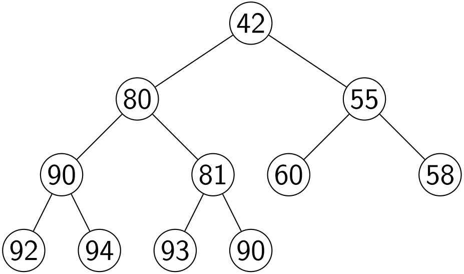


	42	80	55	90	81	60	58	92	94	93	90
Index:	0	1	2	3	4	5	6	7	8	9	10

linkes Kind von Knoten i: 2i+1

rechtes Kind von Knoten i:

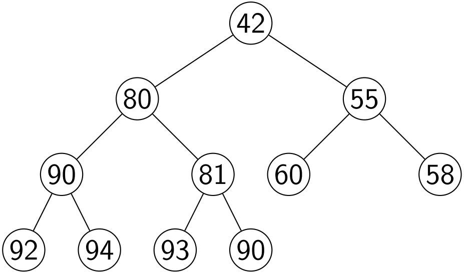
Ein Heap



	42	80	55	90	81	60	58	92	94	93	90
Index:	0	1	2	3	4	5	6	7	8	9	10

linkes Kind von Knoten i: 2i+1
rechtes Kind von Knoten i: 2i+2
Elternknoten von Knoten i:

Ein Heap

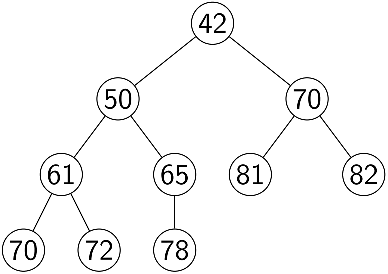


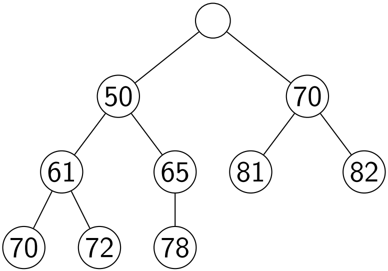
	42	80	55	90	81	60	58	92	94	93	90
Index:	0	1	2	3	4	5	6	7	8	9	10

linkes Kind von Knoten i: $2i+1$
rechtes Kind von Knoten i: $2i+2$
Elternknoten von Knoten i: $(i-1)//2$

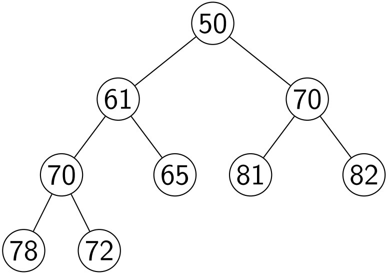
Idee für HeapSort:

Gegeben Liste $[a_0, a_1, ..., a_{n-1}]$
Konstruiere daraus einen Heap.
for i in range(n):
    liefere Wurzel als aktuelles Minimum;
    entferne Wurzel;
    reorganisiere Heap;



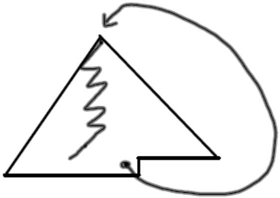


42



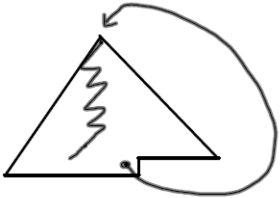
Das Ergebnis ist ein Heap

Aufwand für Reorganisation

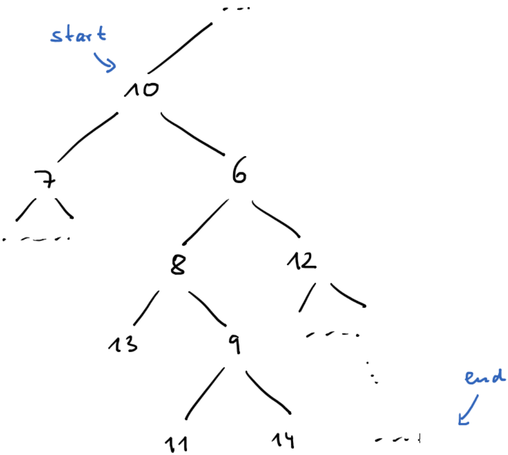


proportional zur Länge des Wegs:

Aufwand für Reorganisation



proportional zur Länge des Wegs: $O(\log n)$



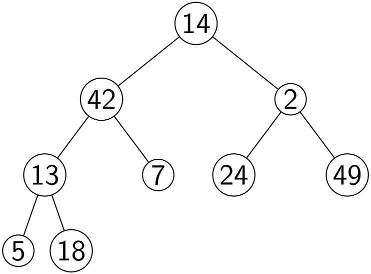
Die Funktion `sift` prüft, ob zwischen `start` und `end` ein ordentlicher Heap ist, oder ob `start` versickern muss. Die Funktion kann sich darauf verlassen, dass unterhalb von `start` alles in Ordnung ist.

```
def sift(a, start , end):
    i = start
    x = a[start]
    j = 2 * i + 1
    if j < end and a[j+1] < a[j]:
        j+=1
    while j <= end and a[j] < x:
        a[i] = a[j]
        i = j
        j = j * 2 + 1
        if j < end and a[j+1] < a[j]:
            j+=1
    a[i] = x
```

Aufbau eines Heaps

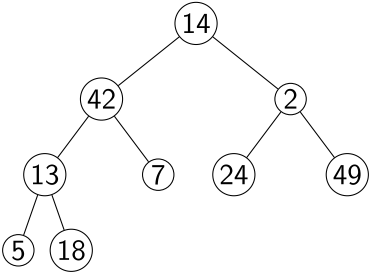
Gegeben Liste a: 14 42 2 13 7 24 49 5 18

Schreibe die Liste als Baum



Aufbau eines Heaps

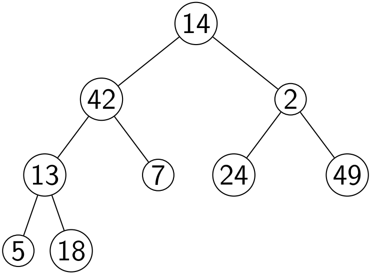
Gegeben Liste a: 14 42 2 13 7 24 49 5 18
Schreibe die Liste als Baum



letzter Knoten:

Aufbau eines Heaps

Gegeben Liste a: 14 42 2 13 7 24 49 5 18
Schreibe die Liste als Baum

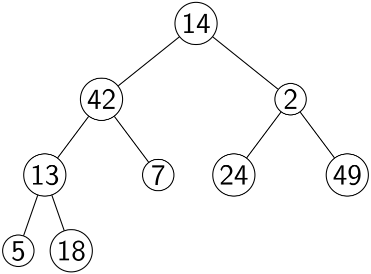


letzter Knoten: a[len(a)-1]

Aufbau eines Heaps

Gegeben Liste a: 14 42 2 13 7 24 49 5 18

Schreibe die Liste als Baum

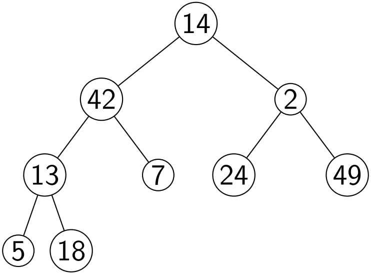


letzter Knoten: a[len(a)-1]

Elternknoten des letzten Knotens:

Aufbau eines Heaps

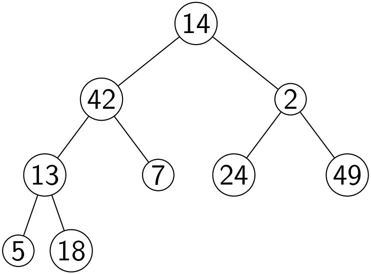
Gegeben Liste a: 14 42 2 13 7 24 49 5 18
Schreibe die Liste als Baum



letzter Knoten: a[len(a)-1]
Elternknoten des letzten Knotens: a[(len(a)-2)//2]

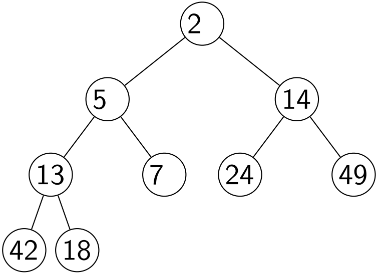
Aufbau eines Heaps

Gegeben Liste a: 14 42 2 13 7 24 49 5 18
Schreibe die Liste als Baum



letzter Knoten: a[len(a)-1]
Elternknoten des letzten Knotens: a[(len(a)-2)//2]

Beginne beim Elternknoten des letzten Knotens. Gehe von dort rückwärts Ebene für Ebene durch und rufe den Elementen jeweils *sift* zu.



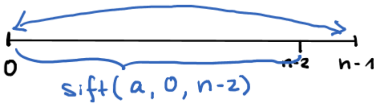
Die Liste nach Aufbau des Heaps:

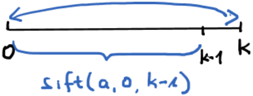
2 5 14 13 7 24 49 42 18

Inhalt der Liste nach jeder Reorganisation:

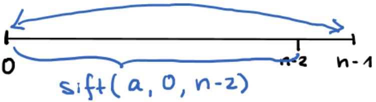
2 5 14 13 7 24 49 42 18
5 7 14 13 18 24 49 42 2
7 13 14 42 18 24 49 5 2
13 18 14 42 49 24 7 5 2
14 18 24 42 49 13 7 5 2
18 42 24 49 14 13 7 5 2
24 42 49 18 14 13 7 5 2
42 49 24 18 14 13 7 5 2
49 42 24 18 14 13 7 5 2

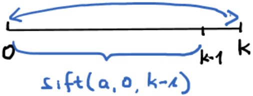
$n = len(a)$





n = len(a)






```
def heap_sort(a):
    n = len(a)
    for k in range ((n-2)//2,-1,-1):
        sift(a,k,n-1)
    for k in range(n-1,0,-1):
        a[0],a[k]=a[k],a[0]
        sift(a,0,k-1)
```

Analyse von HeapSort

Aufwand für Aufbau des Heaps:

Analyse von HeapSort

Aufwand für Aufbau des Heaps: $O(n)$
(Ca. die Hälfte der Elemente des Heaps sind Blätter, nur jeweils ca. n/2 Elemente müssen reorganisiert werden, wobei - nach oben hin - die Anzahl de Elemente mit den längeren Sickerwegen abnimmt.)

Aufwand für eine Reorganisation:

Analyse von HeapSort

Aufwand für Aufbau des Heaps: $O(n)$
(Ca. die Hälfte der Elemente des Heaps sind Blätter, nur jeweils ca. n/2 Elemente müssen reorganisiert werden, wobei - nach oben hin - die Anzahl de Elemente mit den längeren Sickerwegen abnimmt.)

Aufwand für eine Reorganisation: $O(\log n)$

Insgesamt: $O(n \cdot \log n)$ im best, worst und average case.

Zusätzlicher Platzbedarf:

Analyse von HeapSort

Aufwand für Aufbau des Heaps: $O(n)$
(Ca. die Hälfte der Elemente des Heaps sind Blätter, nur jeweils ca. n/2 Elemente müssen reorganisiert werden, wobei - nach oben hin - die Anzahl de Elemente mit den längeren Sickerwegen abnimmt.)

Aufwand für eine Reorganisation: $O(\log n)$

Insgesamt: $O(n \cdot \log n)$ im best, worst und average case.

Zusätzlicher Platzbedarf: $O(1)$

Laufzeit und Platzbedarf der Sortieralgorithmen

	best	average	worst	zusätzlicher Platz
SelectionSort				
BubbleSort				
MergeSort				
QuickSort				
HeapSort				

Laufzeit und Platzbedarf der Sortieralgorithmen

	best	average	worst	zusätzlicher Platz
SelectionSort				
BubbleSort				
MergeSort				
QuickSort				
HeapSort				

Laufzeit und Platzbedarf der Sortieralgorithmen

	best	average	worst	zusätzlicher Platz
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	
BubbleSort				
MergeSort				
QuickSort				
HeapSort				

Laufzeit und Platzbedarf der Sortieralgorithmen

	best	average	worst	zusätzlicher Platz
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
BubbleSort				
MergeSort				
QuickSort				
HeapSort				

Laufzeit und Platzbedarf der Sortieralgorithmen

	best	average	worst	zusätzlicher Platz
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	
MergeSort				
QuickSort				
HeapSort				

Laufzeit und Platzbedarf der Sortieralgorithmen

	best	average	worst	zusätzlicher Platz
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
MergeSort				
QuickSort				
HeapSort				

Laufzeit und Platzbedarf der Sortieralgorithmen

	best	average	worst	zusätzlicher Platz
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
MergeSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	
QuickSort				
HeapSort				

Laufzeit und Platzbedarf der Sortieralgorithmen

	best	average	worst	zusätzlicher Platz
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
MergeSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$
QuickSort				
HeapSort				

Laufzeit und Platzbedarf der Sortieralgorithmen

	best	average	worst	zusätzlicher Platz
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
MergeSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$
QuickSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n^2)$	
HeapSort				

Laufzeit und Platzbedarf der Sortieralgorithmen

	best	average	worst	zusätzlicher Platz
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
MergeSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$
QuickSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n^2)$	$O(\log n)$
HeapSort				

Laufzeit und Platzbedarf der Sortieralgorithmen

	best	average	worst	zusätzlicher Platz
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
MergeSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$
QuickSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n^2)$	$O(\log n)$
HeapSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	

Laufzeit und Platzbedarf der Sortieralgorithmen

	best	average	worst	zusätzlicher Platz
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
MergeSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$
QuickSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n^2)$	$O(\log n)$
HeapSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(1)$