Bäume und Heaps

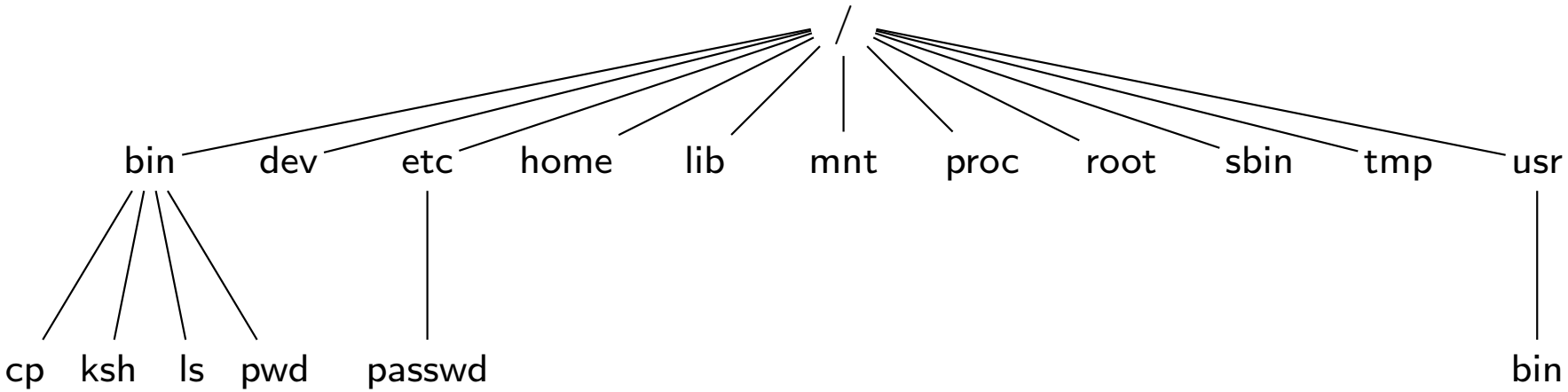
Theorie

Überblick

Bäume stellen eine fundamentale Abstraktion in der Informatik dar und eignen sich dazu, hierachische Strukturen abzubilden.

Informatiker stellen die Wurzel eines Baums oben und die Blätter unten dar!

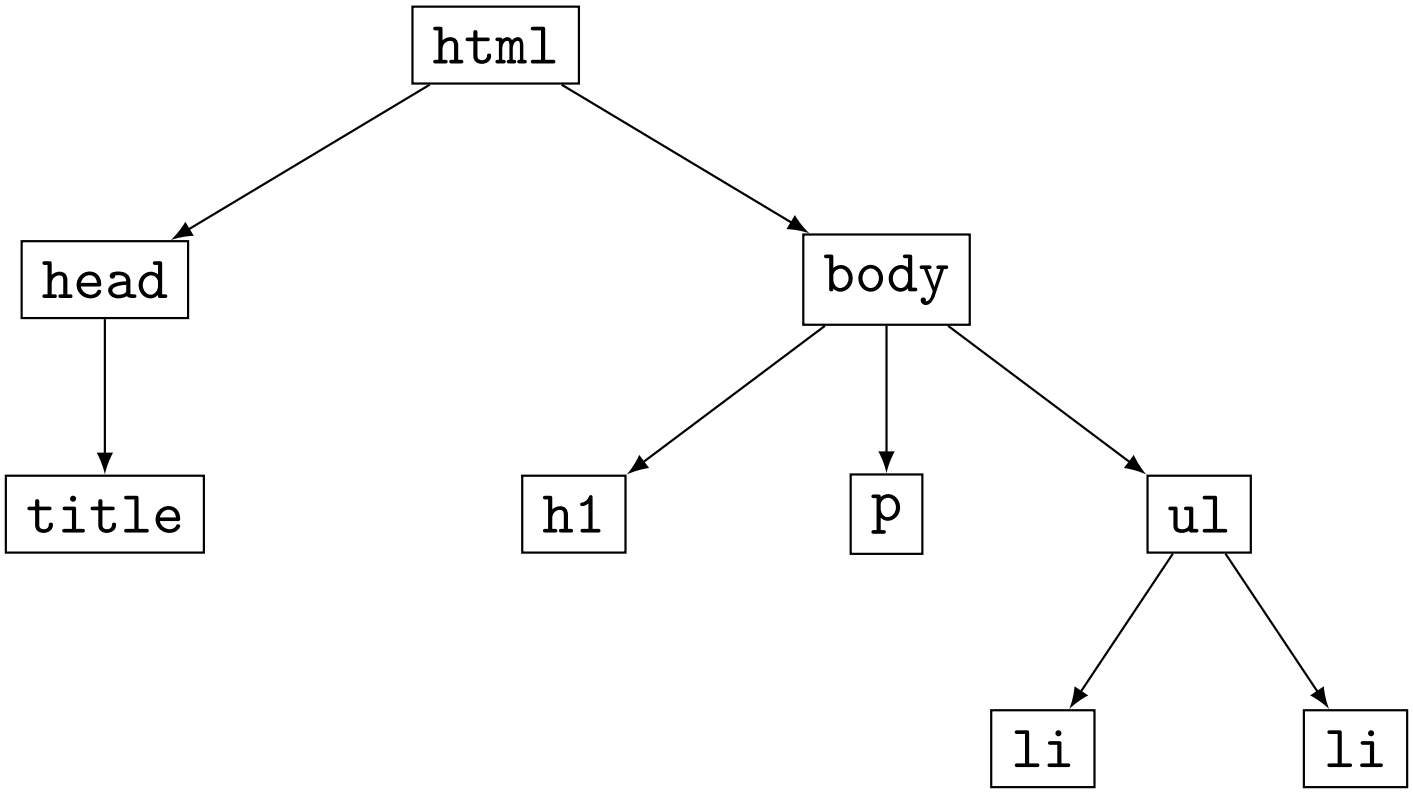
Beispiel: Unix-Dateisystem



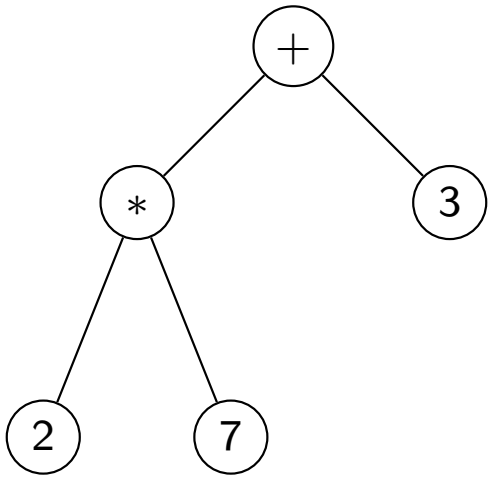
Andere Betriebssysteme haben ebenfalls eine mehr oder weniger feste Verzeichnisstruktur.

HTML-Dokumente

```
1  <html>
2    <head>
3      <title>Simple HTML-Document</title>
4    </head>
5    <body>
6      <h1>Caption</h1>
7      <p>Paragraph</p>
8      <ul>
9        <li>Item 1</li>
10       <li>Item 2</li>
11     </ul>
12   </body>
13 </html>
```


html
head
body
title
h1
p
ul
li
li

Syntaxbäume (Parse Trees)



Begriffe

Knoten (node): Baustein eines Baums, der die Daten enthält.

Kante (edge): Eine Kante verbindet zwei Knoten.

Wurzel (root): Der oberste Knoten eines Baums.

Kind (child): Ein Knoten, der durch eine Kante mit dem unmittelbar darüber liegenden Knoten verbunden ist.

Eltern (parent): Ein Knoten, der durch eine Kante mit einem unmittelbar darunter liegenden Knoten verbunden ist.

Geschwister (sibling): Menge der Knoten, die Kinder des gleichen Elternknotens sind.

Pfad (path): Eine geordnete Folge von Knoten, die durch Kanten verbunden sind.

Blatt (leaf node): Ein Knoten, der keine Kindknoten hat.

Innerer Knoten (interior node): Ein Knoten, der mindestens ein Kind hat.

Teilbaum (subtree): Die Menge der Knoten und Kanten, die aus einem Elternknoten und all seinen Kindern und Kindeskindern besteht.

Tiefe eines Knotens (depth, level): Die Anzahl der Kanten auf dem Pfad vom Wurzelknoten zum betreffenden Knoten. Der Wurzelknoten hat die Tiefe 0.

Höhe eines Knotens (height): Die Anzahl der Kanten entlang des längsten Pfads vom Knoten zu einem Blatt, das im Teilbaum des Knotens liegt.

Höhe eines Baums (height of a tree): Die Höhe des Wurzelknotens oder die maximale Tiefe eines Knotens bzw. Blatts.

Binärbaum (binary tree): Ein Baum, dessen Knoten maximal zwei Kinder haben.

Wald (forest): Eine Menge von Bäumen.

Definition eines Baums (rekursiv)

Ein Baum ist entweder leer oder besteht aus einer Wurzel und null oder mehr Teilbäumen, die jeweils selber ein Baum sind. Die Wurzel jedes Teilbaums ist mit der Wurzel des Elternbaums durch eine Kante verbunden.

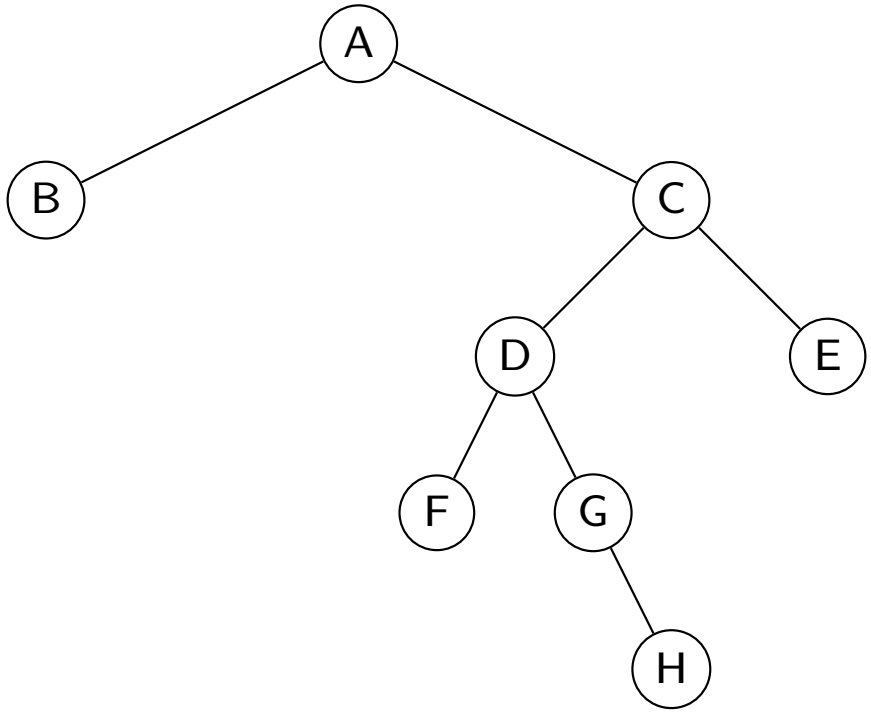
Bemerkung

In der Informatik haben Bäume normalerweise *gerichtete* Kanten. In der Graphentheorie (einem Teilgebiet der Mathematik) können Bäume aber auch ungerichtet sein.

Typologie

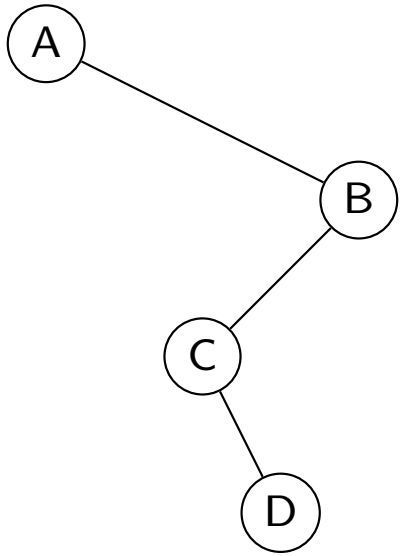
Wir stellen im Folgenden einige häufig vorkommende spezielle Baumstrukturen vor.

Binärbaum (*binary tree*)



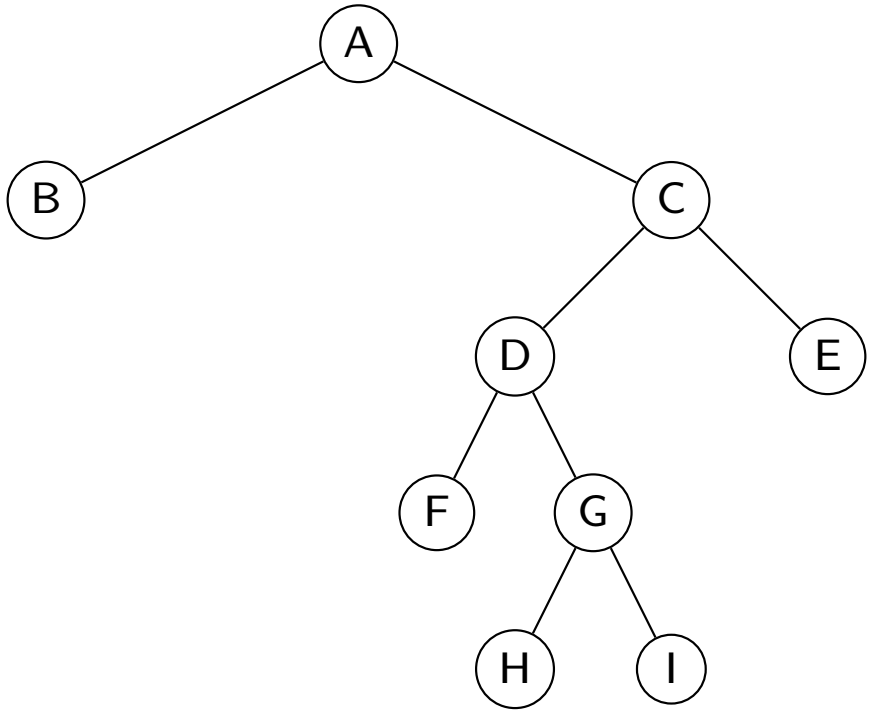
Ein Baum, bei dem jeder innere Knoten maximal zwei Kinder hat.

Entarteter Binärbaum (*degenerated binary tree*)



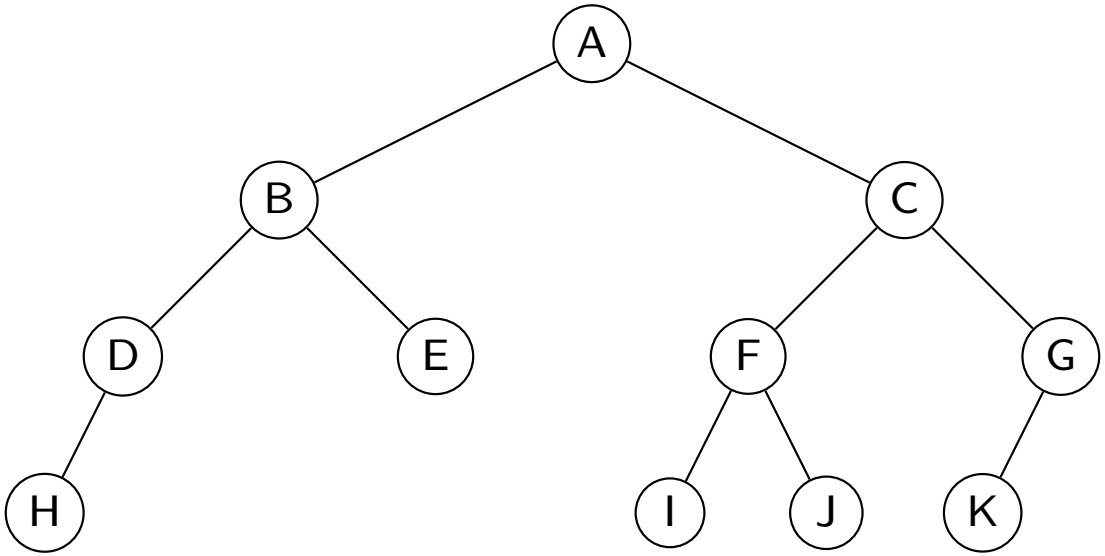
Bei einem entarteteten Baum hat jeder innere Knoten genau ein Kind. Also handelt es sich um einen „Unärbaum“; d. h. um eine lineare Datenstruktur.

Voller Binärbaum (*full binary tree*)



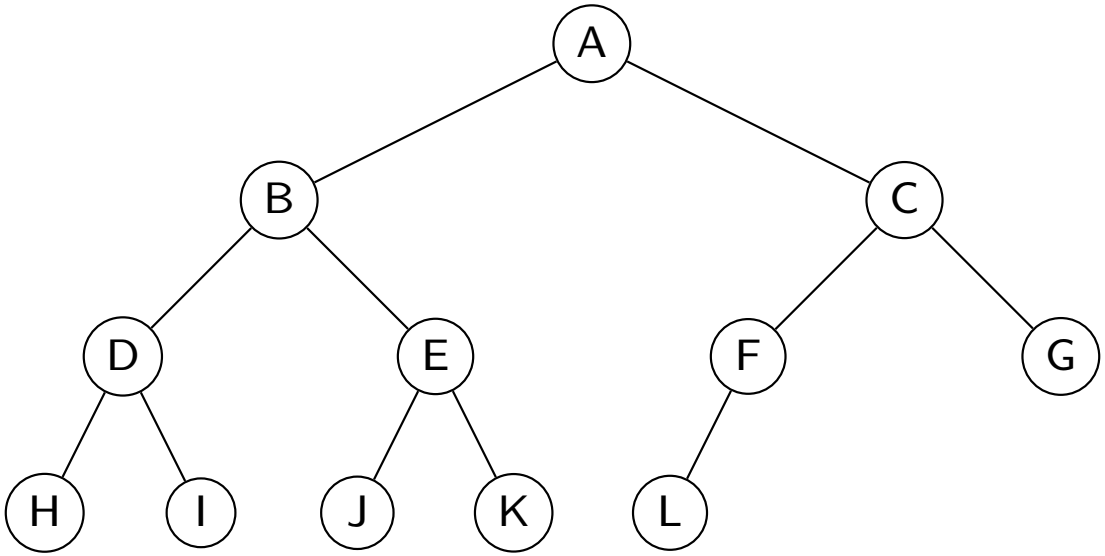
Jeder innere Knoten hat entweder null oder zwei Blätter. Das Adjektiv „voll“ bezieht sich darauf, dass alle inneren Knoten mit zwei Knoten oder Blättern „aufgefüllt“ sind.

Balancierter Binärbaum (*balanced binary tree*)



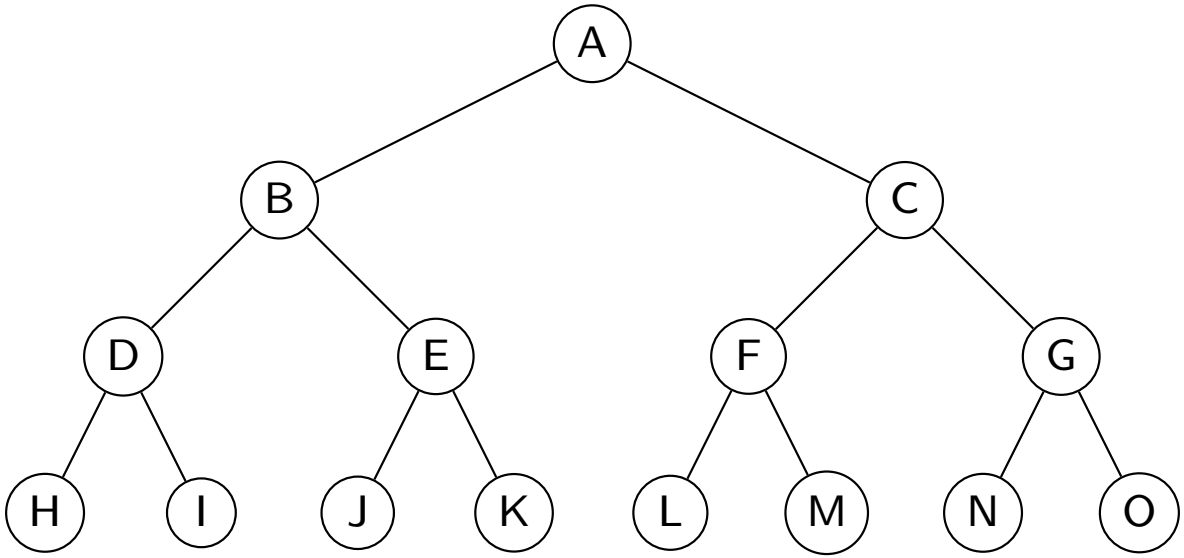
Ein balancierter Binärbaum ist ein Binärbaum, bei dem sich die Tiefe der Blätter höchstens um 1 unterscheiden.

Vollständiger Binärbaum (*complete binary tree*)



Ein vollständiger Binärbaum ist ein balancierter Binärbaum, bei dem alle tieferen Blätter so weit links wie möglich stehen.

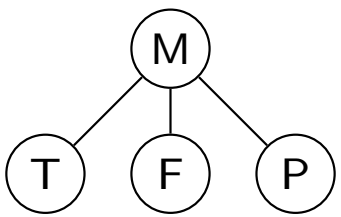
Perfekter Binärbaum (*perfect binary tree*)



Ein perfekter Binärbaum ist ein voller Binärbaum, dessen Blätter dieselbe Tiefe haben.

n-äre Bäume

Entsprechende Baumstrukturen können auch für mehr als zwei Verzweigungen definiert werden. Im Fall von drei Verzweigungen spricht man von ternerären Bäumen (*ternary trees*); im Fall von *n* Verzweigungen spricht man von *n*-ären Bäumen (*n-ary trees*).



Bäumen als verschachtelte Listen

```
tree=['U',['X',['A',[],['S',[],[]]],[]],['M',[],[]]]
```

Repräsentation als Baum:

Bäumen als verschachtelte Listen

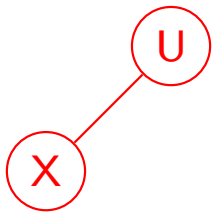
```
tree=['U',['X',['A',[],['S',[],[]]],[]],['M',[],[]]]
```

Repräsentation als Baum:

Bäumen als verschachtelte Listen

```
tree=['U',['X',['A',[],['S',[],[]]],[]],['M',[],[]]]
```

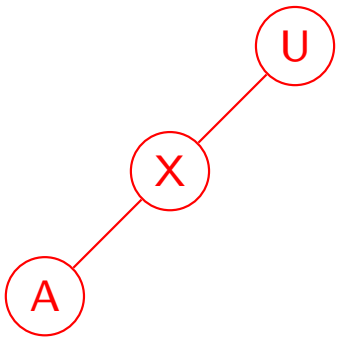
Repräsentation als Baum:



Bäumen als verschachtelte Listen

```
tree=['U',['X',['A',[],['S',[],[]]],[]],['M',[],[]]]
```

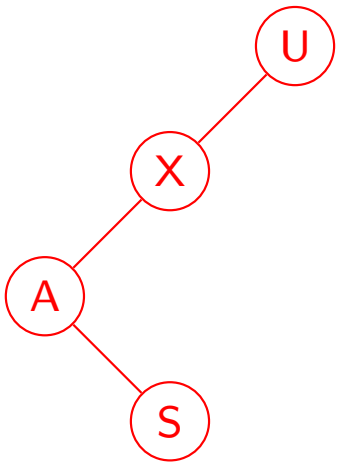
Repräsentation als Baum:



Bäumen als verschachtelte Listen

```
tree=['U',['X',['A',[],['S',[],[]]],[]],['M',[],[]]]
```

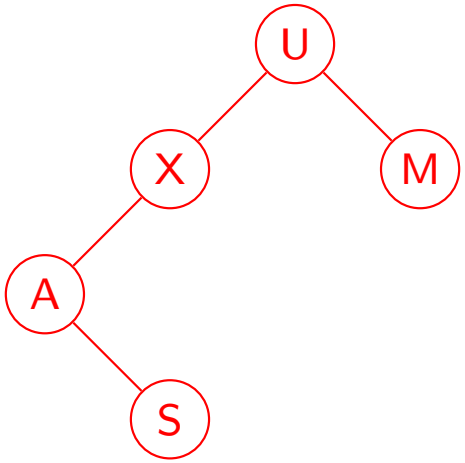
Repräsentation als Baum:



Bäumen als verschachtelte Listen

```
tree=['U',['X',['A',[],['S',[],[]]],[]],['M',[],[]]]
```

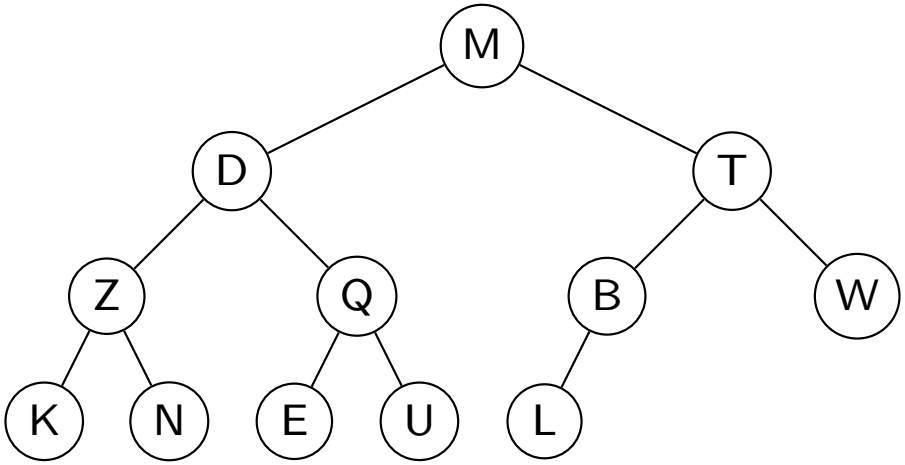
Repräsentation als Baum:

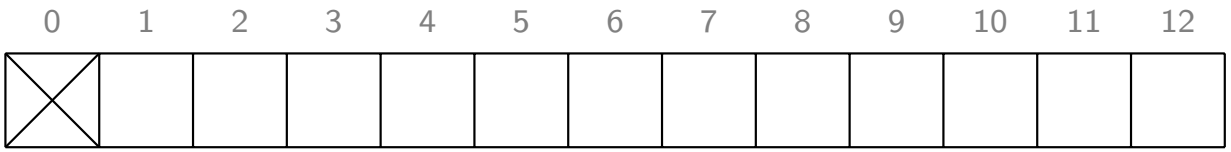


- Jeder Subbaum besteht aus drei Teilen: Wurzel, linker und rechter Knoten
- Fehlende Kinder werden durch leere Listen gekennzeichnet.

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



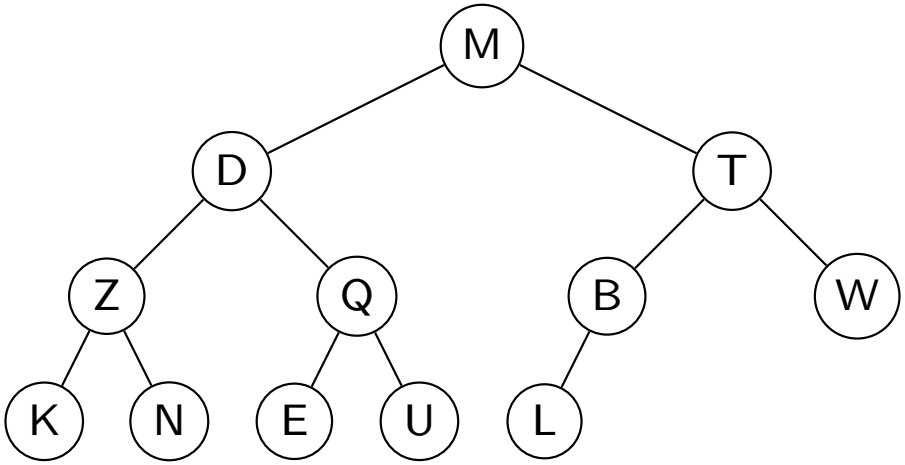


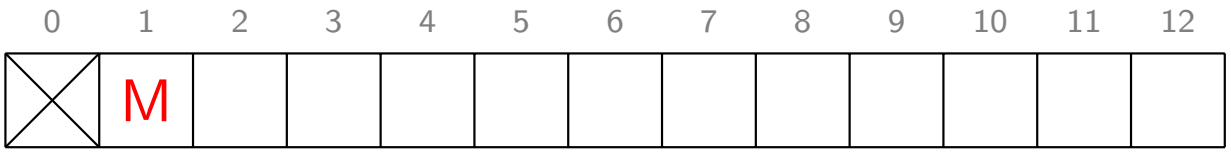
Elternknoten des Kindknotens mit Index i:

linker Kindknoten des Elternknotens mit Index i:

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



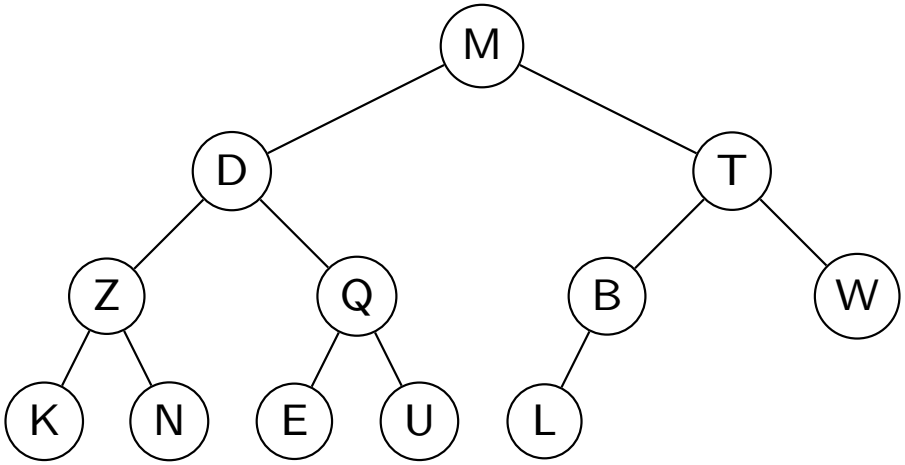


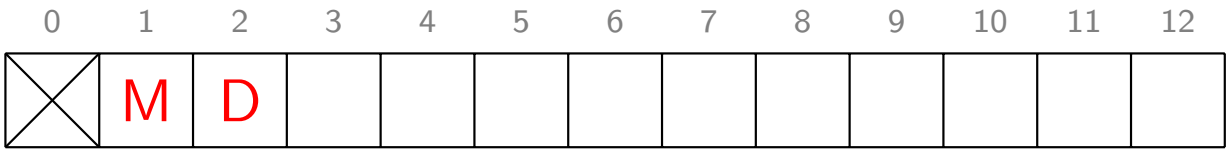
Elternknoten des Kindknotens mit Index i:

linker Kindknoten des Elternknotens mit Index i:

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



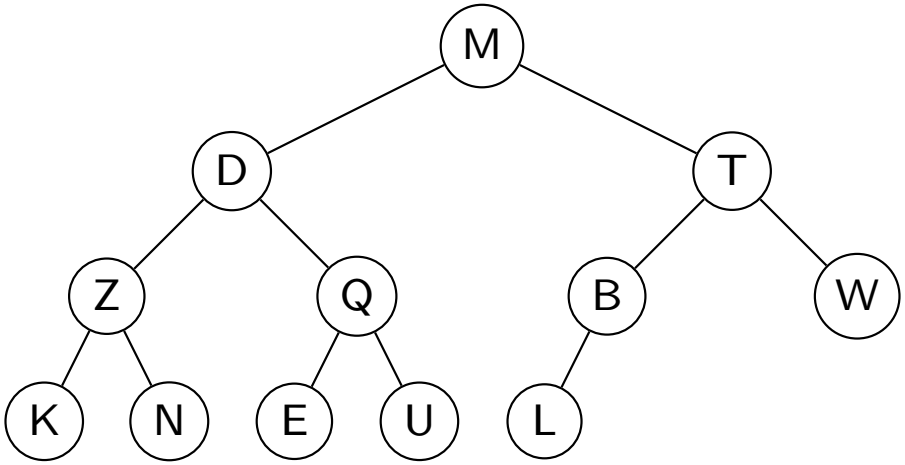


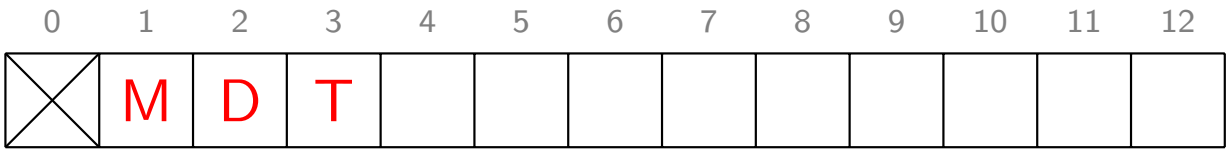
Elternknoten des Kindknotens mit Index i:

linker Kindknoten des Elternknotens mit Index i:

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



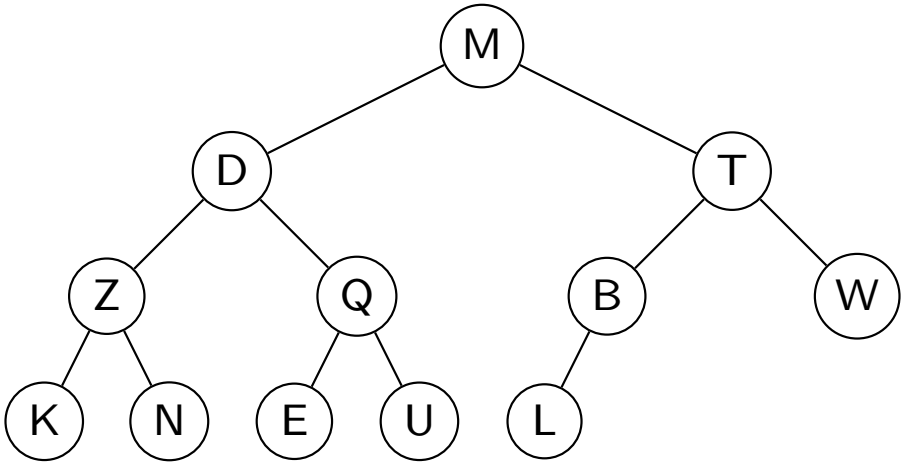


Elternknoten des Kindknotens mit Index i:

linker Kindknoten des Elternknotens mit Index i:

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



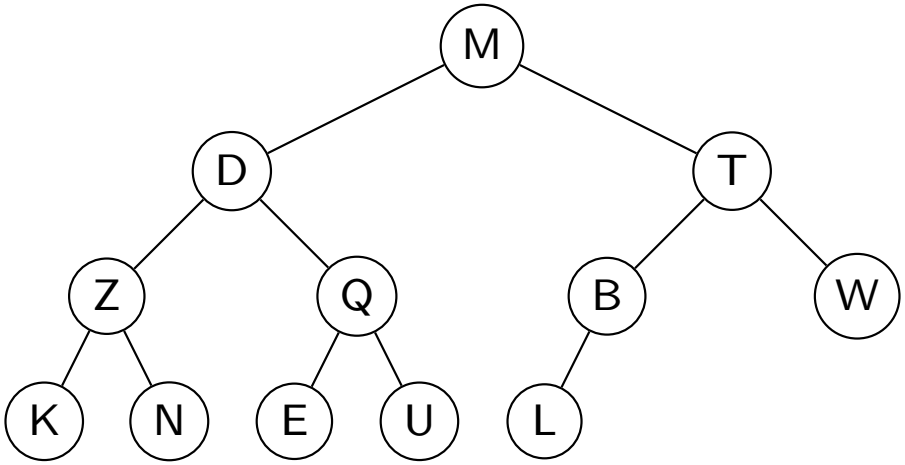
0	1	2	3	4	5	6	7	8	9	10	11	12
⊠	M	D	T	Z								

Elternknoten des Kindknotens mit Index i:

linker Kindknoten des Elternknotens mit Index i:

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



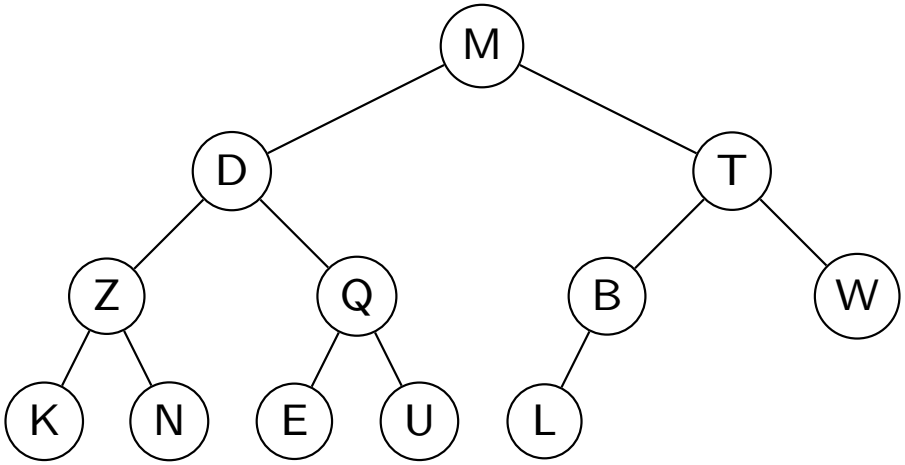
0	1	2	3	4	5	6	7	8	9	10	11	12
⊠	M	D	T	Z	Q							

Elternknoten des Kindknotens mit Index i:

linker Kindknoten des Elternknotens mit Index i:

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



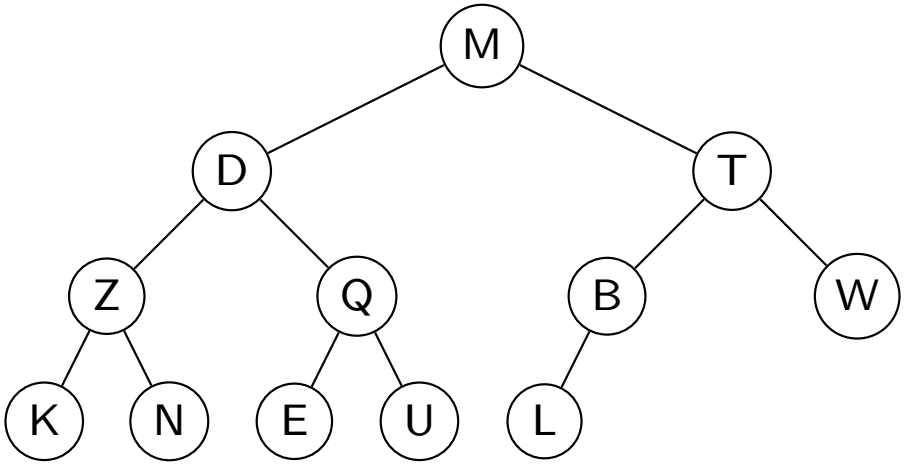
0	1	2	3	4	5	6	7	8	9	10	11	12
⊠	M	D	T	Z	Q	B						

Elternknoten des Kindknotens mit Index i:

linker Kindknoten des Elternknotens mit Index i:

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



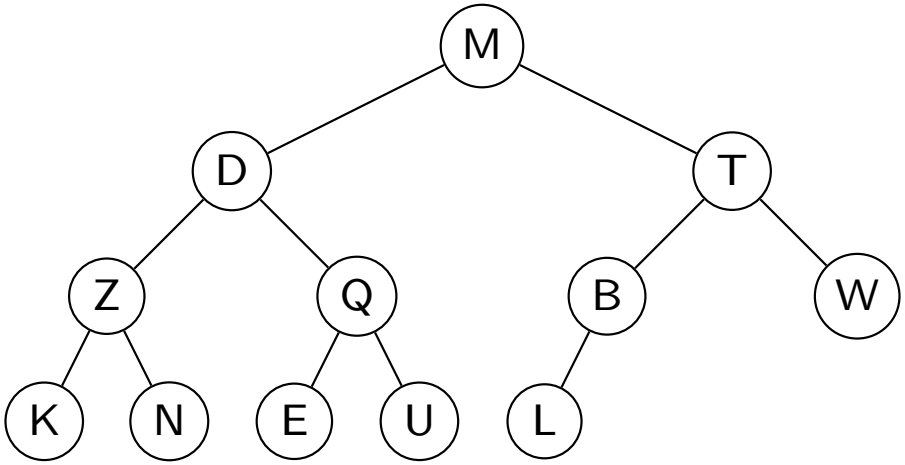
0	1	2	3	4	5	6	7	8	9	10	11	12
	M	D	T	Z	Q	B	W					

Elternknoten des Kindknotens mit Index i:

linker Kindknoten des Elternknotens mit Index i:

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



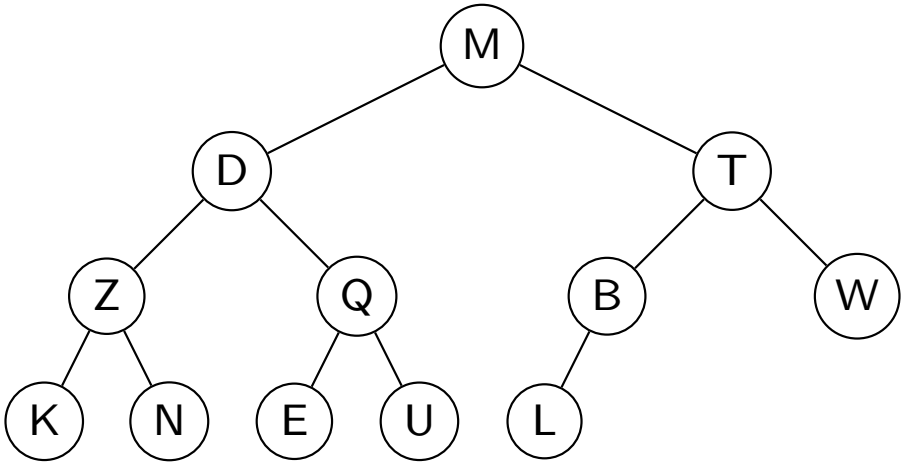
0	1	2	3	4	5	6	7	8	9	10	11	12
	M	D	T	Z	Q	B	W	K				

Elternknoten des Kindknotens mit Index i:

linker Kindknoten des Elternknotens mit Index i:

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



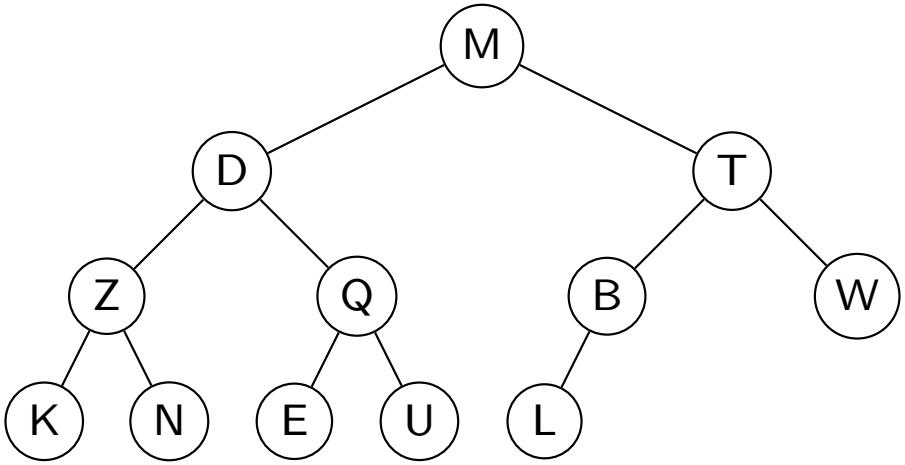
0	1	2	3	4	5	6	7	8	9	10	11	12
	M	D	T	Z	Q	B	W	K	N			

Elternknoten des Kindknotens mit Index i:

linker Kindknoten des Elternknotens mit Index i:

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



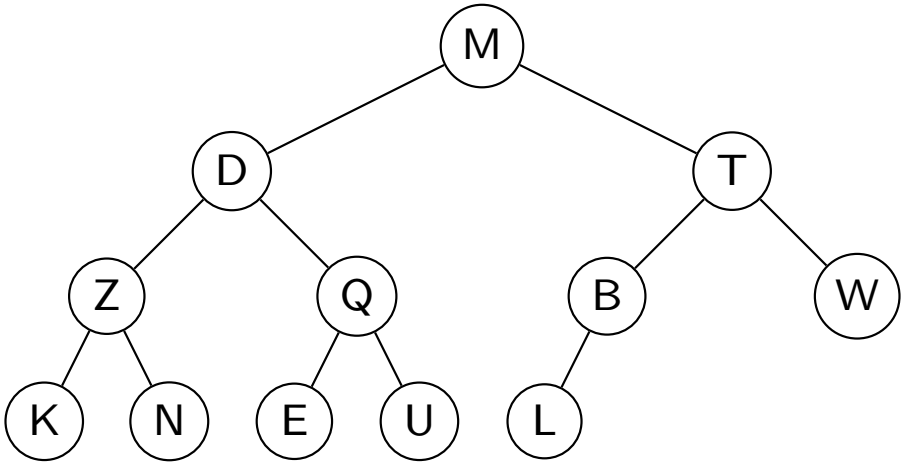
0	1	2	3	4	5	6	7	8	9	10	11	12
	M	D	T	Z	Q	B	W	K	N	E		

Elternknoten des Kindknotens mit Index i:

linker Kindknoten des Elternknotens mit Index i:

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



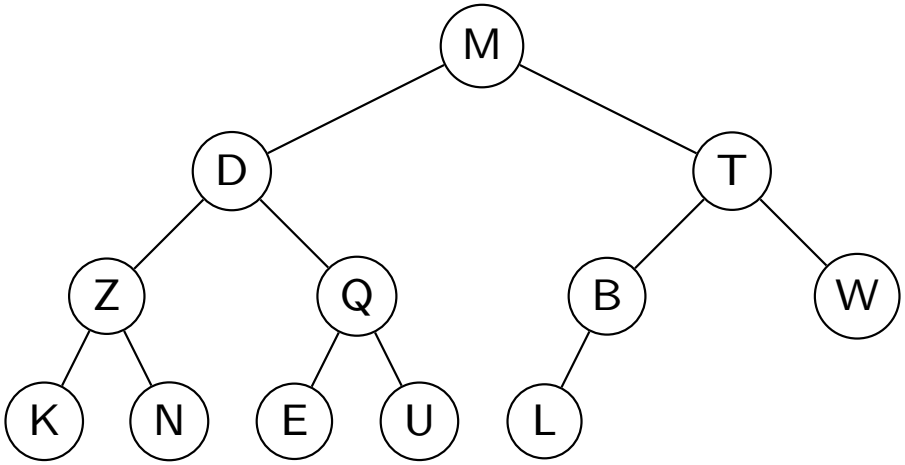
0	1	2	3	4	5	6	7	8	9	10	11	12
	M	D	T	Z	Q	B	W	K	N	E	U	

Elternknoten des Kindknotens mit Index i:

linker Kindknoten des Elternknotens mit Index i:

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



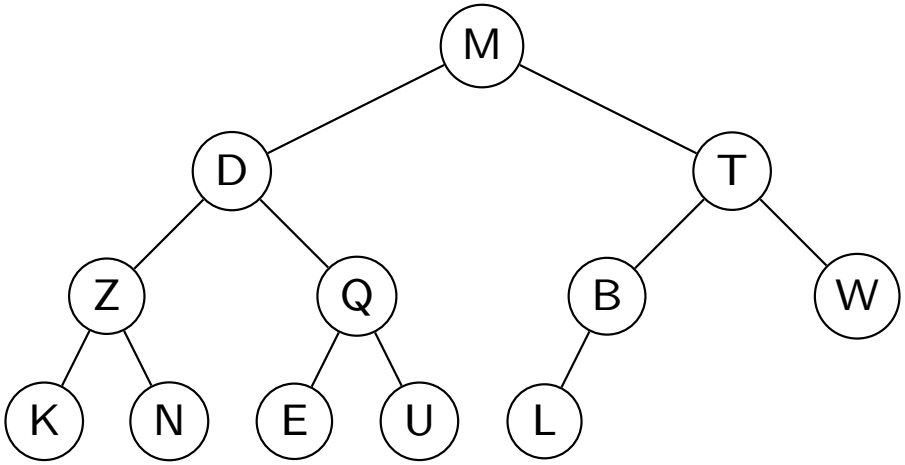
0	1	2	3	4	5	6	7	8	9	10	11	12
	M	D	T	Z	Q	B	W	K	N	E	U	L

Elternknoten des Kindknotens mit Index i:

linker Kindknoten des Elternknotens mit Index i:

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



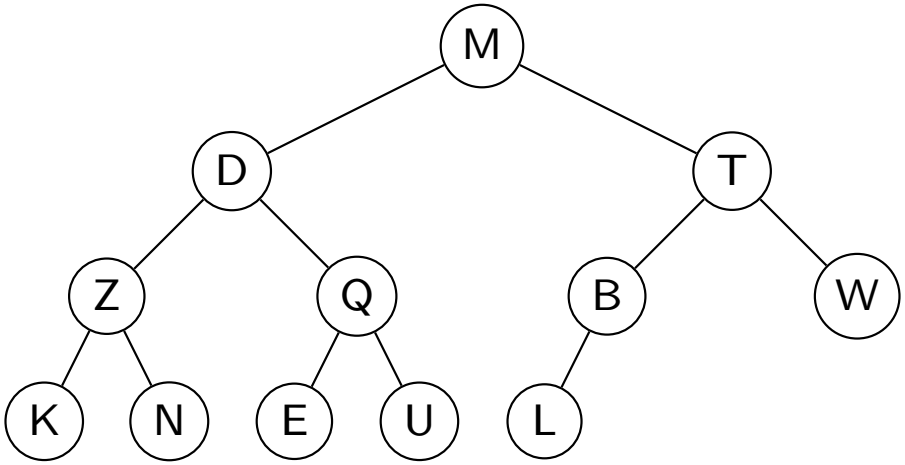
0	1	2	3	4	5	6	7	8	9	10	11	12
	M	D	T	Z	Q	B	W	K	N	E	U	L

Elternknoten des Kindknotens mit Index i:

linker Kindknoten des Elternknotens mit Index i:

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



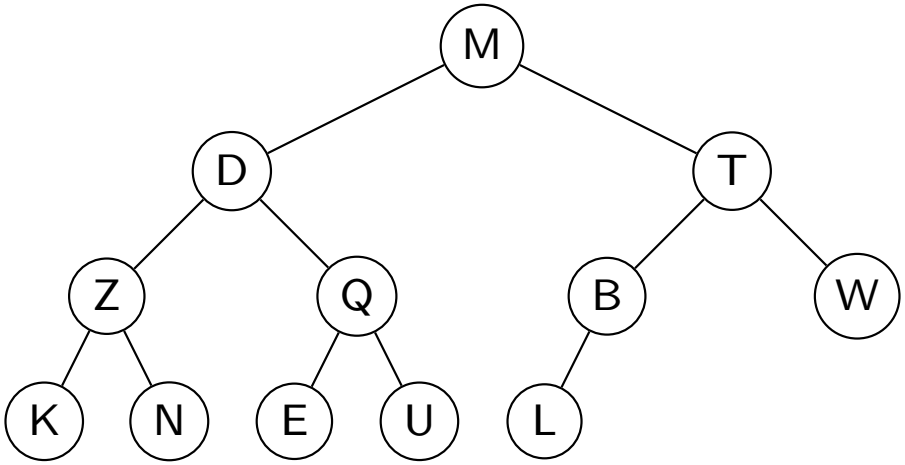
0	1	2	3	4	5	6	7	8	9	10	11	12
	M	D	T	Z	Q	B	W	K	N	E	U	L

Elternknoten des Kindknotens mit Index i: $\lfloor i/2 \rfloor$

linker Kindknoten des Elternknotens mit Index i:

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



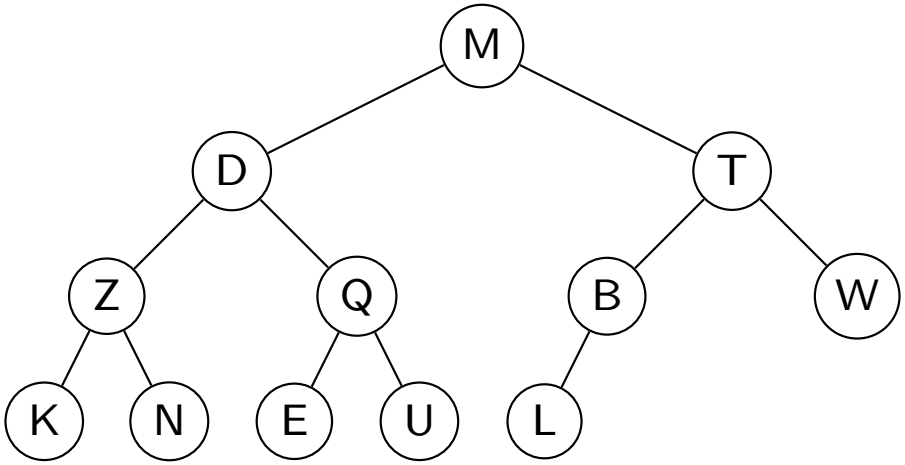
0	1	2	3	4	5	6	7	8	9	10	11	12
	M	D	T	Z	Q	B	W	K	N	E	U	L

Elternknoten des Kindknotens mit Index i: $\lfloor i/2 \rfloor$

linker Kindknoten des Elternknotens mit Index i: $2i$

Heaps

Heap: Datenstrukur, die einen vollständigen Binärbaum als Liste speichert.



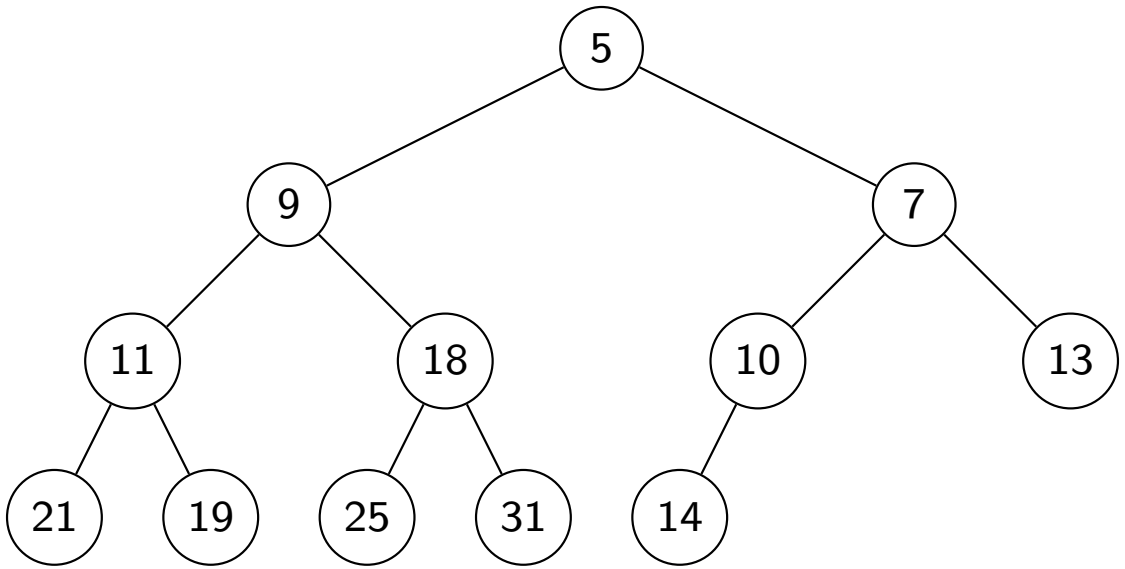
0	1	2	3	4	5	6	7	8	9	10	11	12
	M	D	T	Z	Q	B	W	K	N	E	U	L

Elternknoten des Kindknotens mit Index i: $\lfloor i/2 \rfloor$

linker Kindknoten des Elternknotens mit Index i: $2i$

Min- und Max-Heaps

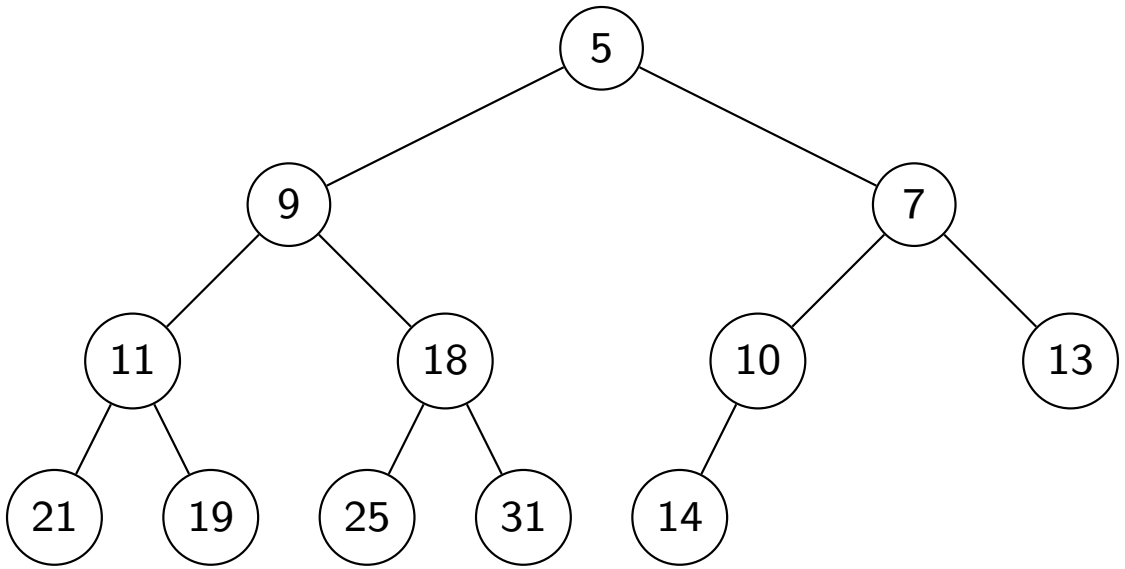
Ein Min-Heap ist ein Heap, bei dem jeder Kindknoten einen Schlüssel hat, der grösser (oder gleich) wie der seines Elternknotens ist. Max-Heaps werden analog definiert.



Min-Heap-Eigenschaft:

Min- und Max-Heaps

Ein Min-Heap ist ein Heap, bei dem jeder Kindknoten einen Schlüssel hat, der grösser (oder gleich) wie der seines Elternknotens ist. Max-Heaps werden analog definiert.

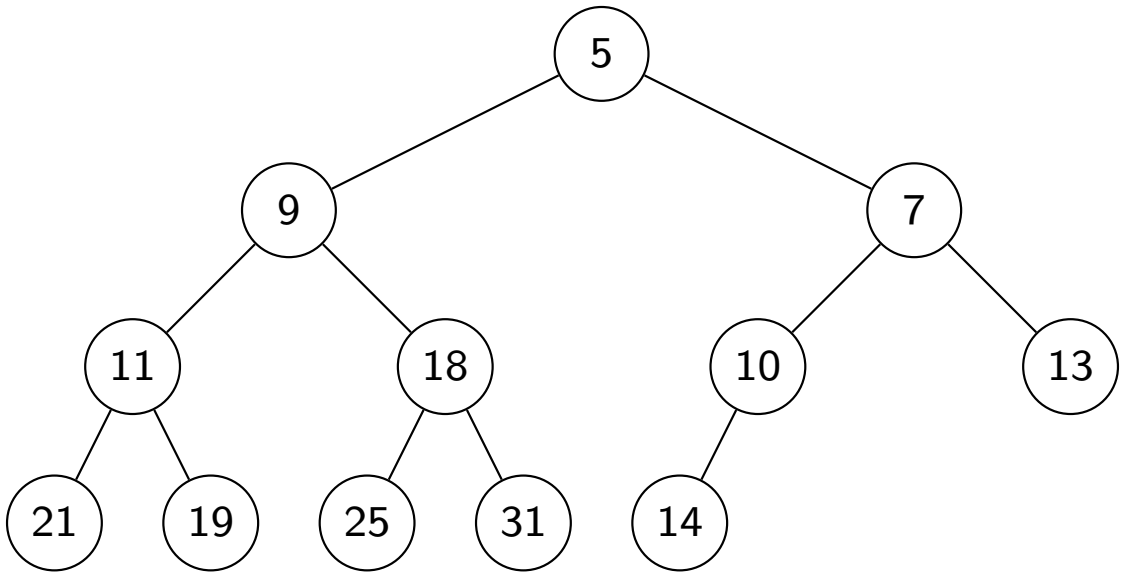


Min-Heap-Eigenschaft: Der kleinste Schlüssel ist in der Wurzel.

```
1  class MinHeap:
2
3      def __init__(self):
4          self.heap = [None]
5          self.size = 0
6
7      def __str__(self):
8          return str(self.heap)
9
10     def swap(self, i, j):
11         self.heap[i],self.heap[j] =
               self.heap[j],self.heap[i]
```

Schlüssel hinzufügen

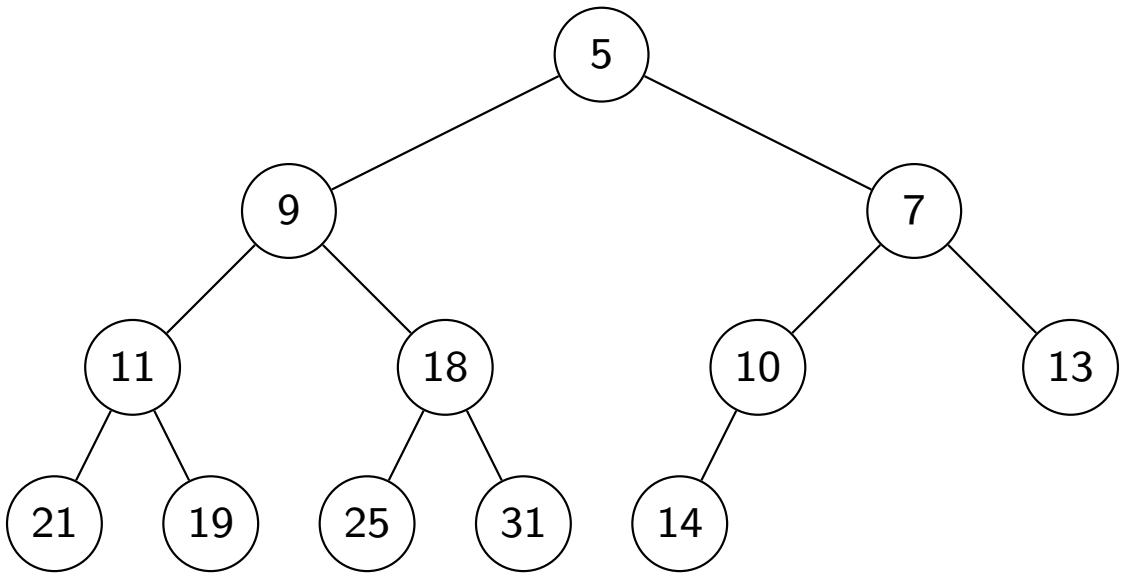
Ein neuer Schlüssel (3) wird an die letzte Position des Heaps gesetzt und so lange „hochgetrieben“ (`swim`), bis die Min-Heap-Eigenschaft wieder hergestellt ist.



```
13      def insert(self, item):
14          self.heap.append(item)
15          self.size += 1
16          self.swim(self.size)
17
18      def swim(self, i):
19          while i // 2 > 0:
20              if self.heap[i] < self.heap[i//2]:
21                  self.swap(i, i//2)
22              i = i // 2
```

Schlüssel an der Wurzel entfernen

Wird ein Schlüssel an der Wurzel des Baums entfernt, so rückt der letzte Schlüssel im Baum (Heap) an diese Position nach. Danach wird dieser Schlüssel so lange „heruntergetrieben“ (`sink`), bis die Min-Heap-Eigenschaft wieder hergestellt ist.

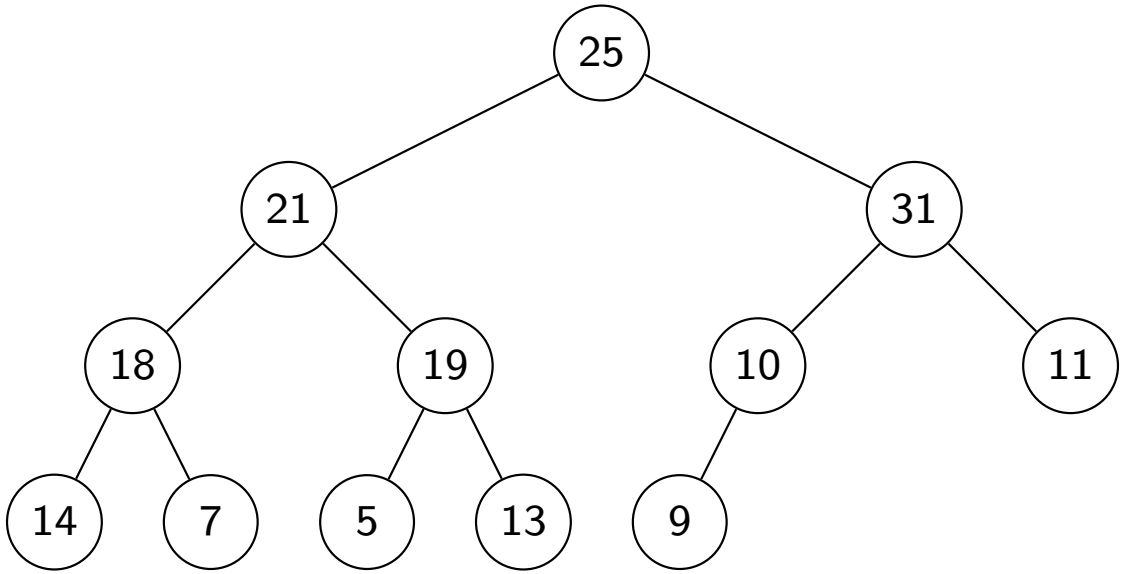


```
24      def delMin(self):
25          top = self.heap[1]
26          self.heap[1] = self.heap[self.size]
27          self.size -= 1
28          self.heap.pop()
29          self.sink(1)
30          return top
31
32      def sink(self, i):
33          while 2*i <= self.size:
34              j = self.minChild(i)
35              if self.heap[i] > self.heap[j]:
36                  self.swap(i, j)
37              i = j
```

```
38
39      def minChild(self, i):
40          # falls der letzte Elternknoten nur ein Kind hat:
41          if 2*i+1 > self.size:
42              return 2*i
43          else:
44              if self.heap[2*i] < self.heap[2*i+1]:
45                  return 2*i
46              else:
47                  return 2*i+1
```

Einen Heap in einen Min-Heap verwandeln

Es genügt, alle Schlüssel mit einem Index kleiner als $\lfloor n/2 \rfloor$ (innere Knoten) so weit wie nötig „sinken“ zu lassen.



```
48
49      def buildHeap(self, L):
50          self.heap = [None] + L[:]
51          self.size = len(L)
52          i = self.size//2
53          while i > 0:
54              self.sink(i)
55              i = i-1
```