Datenstrukturen

Theorie

Version vom 26. August 2019

1 Datentypen

Der Datentyp besagt ...

- welchen Wertebereich die Daten haben,
- welche Operationen auf den Daten zulässig sind.

1.1 Einfache Datentypen

Ein einfacher Datentyp (Synonyme: elementarer oder primitiver Datentyp) besteht aus einem einzelnen Wert.

In Python sind das (unter anderem):

- ganze Zahlen (integer): `+`, `-`, `*`, `//`, ...
- Gleitkommazahlen (float): `+`, `-`, `*`, `/`, ...
- Wahrheitswerte (boolean): `not`, `and`, `or`

1.2 Zusammengesetzte Datentypen

Zusammengesetzte Datentypen sind Zusammenfassungen (Datenstrukturen) aus einfachen Datentypen.

Im folgenden werden einige der in Python bereits vorhandenen zusammengesetzten Datentypen wiederholt.

Listen

Eine Liste ist eine geordnete Menge von null oder mehr Referenzen auf Python-Daten.

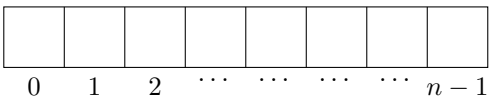


Abbildung 1.1: Modell einer Liste

Operation	Effizienz
`L[i]`	$O(1)$
`L[3]=25`	$O(1)$
`L.append(item)`	$O(1)$
`L.pop()`	$O(1)$
`L.pop(i)`	$O(n)$
`L[i:j]`	$O(k)$
`L1+L2`	$O(k)$
`L.del(i)`	$O(n)$
`L.insert(i,item)`	$O(n)$
`x in L`	$O(n)$
`for item in L`	$O(n)$
`L.reverse`	$O(n)$
`L.sort()`	$O(n \log n)$

Tabelle 1.1: Effizienz einiger Funktionen und Methoden für Listen

Tupel

Tupel sind unveränderliche Listen. Das heisst, dass von den Listenmethoden nur diejenigen anwendbar sind, welche das Tupel und ihre Elemente nicht modifizieren.

Bei der literalen Eingabe werden Tupel durch runde anstelle von eckigen Klammern definiert.

Zeichenketten

Zeichenketten (strings) sind Folgen aus null oder mehr Symbolen (Zeichen).

Bei der literalen Eingabe werden Zeichenketten durch einfache oder doppelte Anführungszeichen gekennzeichnet.

Strings verhalten sich wie Tupel, deren Elemente die einzelnen Symbole sind. Die Methoden zum Sortieren, Löschen oder Einfügen sind daher nicht unmittelbar anwendbar.

Sets

Eine Menge ist eine ungeordnete Sammlung unveränderlicher Python-Objekte ohne Duplikate. Die Mengen werden literal als komma-separierte Listen dargestellt, die von geschweiften Klammern eingeschlossen sind.

`A = {1, 'Hallo', 3.14, True}`

`B = set()` (die leere Menge)

Beispiel	Erklärung
`x in M`	Ist `x` ein Element von `M`?
`len(M)`	Anzahl der Elemente in der Menge `M`
`A | B`	Vereinigungsmenge $A \cup B$
`A & B`	Schnittmenge $A \cap B$
`A - B`	Mengendifferenz $A \setminus B$
`A <= B`	Teilmenge $A \subset B$?

Tabelle 1.2: Operationen für Sets

Beispiel	Erklärung
`A.union(B)`	$A \cup B$
`A.intersection(B)`	$A \cap B$
`A.difference(B)`	$A \setminus B$
`A.issubset(B)`	$A \subset B$
`A.add(item)`	Füge `item` zur Menge `A` hinzu.
`A.remove(item)`	Entferne `item` aus Menge `A`.
`A.pop()`	Entferne beliebiges Element aus Menge `A`.
`A.clear()`	Entferne alle Element aus Menge `A`.

Tabelle 1.3: Methoden für Sets

Dictionaries (Assoziative Listen)

Eine Dictionary ist eine ungeordnete Sammlung von Schlüssel-Wert Paaren (key-value pair).

Dictionaries werden literal als komma-separierte Listen der Schlüssel-Wert-Paare dargestellt, die von geschweiften Klammern eingeschlossen sind. Die Schlüssel-Wert Paare selbst werden durch einen Doppelpunkt getrennt.

Operation	Effizienz
`D[key]`	$O(1)$
`D[key]=25`	$O(1)$
`D.del(key)`	$O(1)$
`key in D`	$O(1)$
`for key in D`	$O(n)$
`D.copy()`	$O(n)$

Tabelle 1.4: Effizienz einiger Funktionen und Methoden für Dictionaries

1.3 Abstrakte Datentypen

Schnittstellen

Autofahrer müssen nicht unbedingt etwas von Motoren, Getriebe, oder Bremsen verstehen, um ein Auto fahren zu können. Gas- und Bremspedal, ein Getriebe und eine Zündung sorgen dafür, dass auch Nicht-Ingenieure ein Auto bedienen können.

Ähnlich verhält es sich beim Informatikanwender. Er muss nichts über Variablen, Schleifen oder Unicode wissen, um Dokumente zu schreiben, Mails zu verschicken oder im Web zu surfen (auch wenn es hilfreich ist).

In beiden Beispielen muss der Benutzer (*Client*) einer Abstraktion nichts von den darunter liegenden Details wissen, so lange er die *Schnittstelle* (Steuerrad, Bremspedal, Browser, Textverarbeitungsprogramm) versteht und bedienen kann.

Im Zusammenhang mit Datentypen wollen wir den Begriff der Schnittstelle noch weiter einschränken:

Unter einer Schnittstelle versteht man eine Beschreibung aller Operationen (Methoden), mit denen auf eine Sammlung von Daten zugegriffen werden kann.

In vielen Fällen wünschen sich Programmierer für eine Aufgabe eine massgeschneiderte Datenstruktur, die das Lösen der Aufgabe möglichst einfach macht.

Daher schafft man sogenannte *Abstrakte Datentypen* (*abstract data type, ADT*), welche eine logische Sicht auf die Daten erlaubt, die nicht unbedingt mit der physikalischen Darstellung der Daten übereinstimmen muss.

Ein abstrakter Datentyp (ADT) ist eine Sammlung von Objekten sowie eine Beschreibung der zulässigen Operationen, die darauf zugreifen.

Kapselung und Information Hiding

Bei der Beschreibung der Operationen eines abstrakten Datentyps sind zwei Dinge wesentlich:

- *Kapselung:* Der Zugriff auf die Operation darf nur über die Abstraktion der Schnittstelle erfolgen.
- *Information Hiding:* Die Details der Implementierung (Variablen) müssen vor dem Client verborgen werden. Bei einer Änderung der Implementierung könnten diesese Details nicht mehr gültig sein und zu Fehlern führen.

Durch stabile Schnittstellen können komplexe Softwaresysteme verbessert werden, ohne das gesamte System verändern zu müssen.

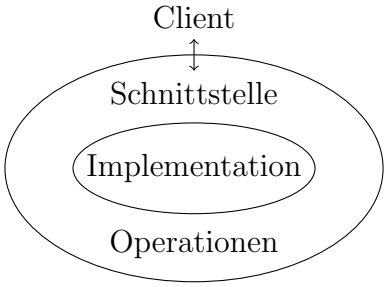
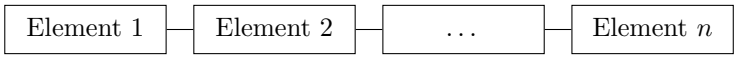


Abbildung 1.2: Abstrakter Datentyp

2 Lineare Datenstrukturen

Überblick

In linearen Datenstrukturen lassen sich die Daten in einer Weise repräsentieren, dass, ausgenommen vom letzten Element, jedes Element einen eindeutiger Nachfolger besitzt.



Die jeweils möglichen Operationen definieren den spezifischen Typ.

2.1 Stacks (Stapel)

- Last In First Out (LIFO)
- Hinzufügen (push) und Entfernen (pop) erfolgen von derselben Seite
- Anwendung: Browser-History; Undo-Funktion von Anwendungsprogammen; Auswertung von Postifix-Ausdrücken und Übersetzung von Infix-Ausdrücken in Postfix-Form; Backtracking-Algorithmen; Verwaltung des Arbeitsspeichers von Computern

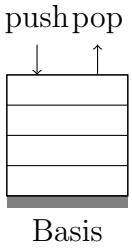


Abbildung 2.1: Modell eines Stacks

Python-Implementation eines Stacks

Die Klasse `Stack` wird als Python-Liste implementiert. Die Methoden sind:

- Der Konstruktor `Stack()` erzeugt einen leeren Stack.
- `push(item)` legt `item` auf den Stack ab.
- `pop()` entfernt das oberste Element vom Stack und liefert es als Wert zurück.
- `peek()` liefert das oberste Element des Stacks als Wert zurück ohne es zu entfernen.
- `isEmpty()` liefert `True` bzw. `False` als Rückgabewert, wenn der Stack leer bzw. nicht-leer ist.
- `s.size()` liefert die Anzahl Elemente des Stacks zurück.

Der Quellcode

```
class Stack:

    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[-1]

    def size(self):
        return len(self.items)
```

Anwendung

Schreibe eine Funktion `check(string)`, das einen String als Argument entgegen nimmt und überprüft, ob die (runden) Klammern in diesem String korrekt gesetzt sind.

Lösungsidee: Erzeuge einen leeren Stack `s`. Durchlaufe die Zeichenkette zeichenweise. Ist das Symbol eine öffnende Klammer, kommt es auf den Stack. Ist das Symbol eine schliessenden Klammer, so entferne das oberste Stackelement. Überprüfe, ob diese Operation auf einem leeren Stack ausgeführt wird. Wenn ja, gibt es mehr schliessende als öffende Klammern und der Ausdruck ist falsch. Nachdem der gesamte String verarbeitet wurde, ist noch zu prüfen, ob noch Klammern auf dem Stack liegen. Wenn ja, dann gibt es mehr öffnende als schliessende Klammern und der Ausdruck ist falsch.

```
from stack import Stack

def check(string):

    s = Stack()

    for symbol in string:
        if symbol == '(':
            s.push(symbol)
        elif symbol == ')':
            if s.isEmpty():
                return False
            else:
                s.pop()
        else: # andere Symbole ignorieren
            pass

    return s.size() == 0 # True, falls Stack leer
```

2.2 Queues (Warteschlangen)

- First In First Out (FIFO)
- Hinzufügen (enqueue) und Entfernen (dequeue) erfolgen an entgegengesetzten Seiten
- Beispiele: Druckerwarteschlangen; Warteschlange von Computerprozessen, Simulationen (Strassenverkehr, Supermarkt, ...)

Abbildung 2.2: Modell einer Queue

Python-Implementation einer Queue

Die Klasse `Queue` soll als Python-Liste implementiert werden. Die Methoden sind:

- Der Konstruktor `Queue()` erzeugt eine leere Queue.
- `enqueue(item)` fügt `item` am Ende der Queue ein.
- `dequeue()` entfernt das vorderste Element der Queue.
- `isEmpty()` liefert `True` bzw. `False` als Rückgabewert, wenn die Queue leer bzw. nicht-leer ist.
- `s.size()` liefert die Anzahl Elemente der Queue zurück.

Anwendung: Gesellschaftsspiel-Simulation

Bei dem in den USA verbreiteten Kinderspiel *Hot Potato* geht es darum, dass die im Kreis sitzenden Teilnehmer einen Gegenstand (die heisse Kartoffel) weiterreichen. Derjenige, der nach Ablauf einer (zufälligen) Frist, den Gegenstand in der Hand hält, scheidet aus.

Schreibe eine Funktion `hotPotatoe(namelist, k)` mit einer Namensliste und einer ganzen Zahl `k` als Argument.

Die Namen werden in eine Warteschlange eingefüllt. In jeder Runde wird die Warteschlange um k Positionen zyklisch vertauscht und derjenige Name, der danach ganz vorne in der Schlange steht, entfernt. Am Ende gibt das Programm den Namen der Person aus, die alle Runden überstanden hat.

```
from random import randint
from myqueue import Queue

L = ['Bill', 'Sue', 'Kent', 'Jane', 'Brad', 'Kim']
q = Queue()

for name in L:
    q.enqueue(name)

while q.size() > 1:
    n = q.size()-1
    # Elemente zyklisch verschieben
    for i in range(0, randint(0, n)):
        q.enqueue(q.dequeue())

    q.dequeue()

print(q)
```

2.3 Deques (zweiseitige Warteschlangen)

Deque [ausgesprochen „Deck“] steht für Double-ended queue.

Abbildung 2.3: Modell einer Deque

- Hinzufügen (add) und Entfernen (remove) erfolgen jeweils an beiden Seiten
- Anwendungen: Textsuche mittels regulärer Ausdrücke, nichtdeterministische endliche Automaten

Python-Implementation einer Deque

Die Klasse `Deque` soll auf der Basis einer Python-Liste implementiert werden. Die Methoden sind:

- Der Konstruktor `Deque()` erzeugt eine leere Deque.
- `addFront(item)` fügt `item` am vorderen Ende ein.
- `addRear(item)` fügt `item` am hinteren Ende ein.
- `removeFront(item)` entfernt `item` vom vorderen Ende.
- `removeRear(item)` entfernt `item` vom hinteren Ende.
- `isEmpty()` liefert `True` bzw. `False`, wenn die Deque leer bzw. nichtleer ist.
- `s.size()` liefert die Anzahl Elemente der Queue zurück.

Anwendung: Palindrome erkennen

- Ein Palindrom-Kandidat wird zeichenweise in eine Deque „geschoben“.
- So lange die Deque mehr als ein Element besitzt, wird von beiden Seiten jeweils ein Element entfernt. Sind die beiden Elemente verschieden, liefert die Funktion den Wert `False` zurück.
- Hat die Deque nur noch ein Element oder ist sie leer, muss es sich um ein Palindrom handeln und die Funktion liefert `True` zurück.

```
from deque import Deque

def palinChecker(text):

    d = Deque()

    for character in text:
        d.addFront(character)

    while d.size() > 1:
        if d.removeFront() != d.removeRear():
            return False

    return True

# Prüfe, ob 11**0, 11**1, 11**2, ... 11**6 Palindrome sind:
for i in range(0, 6):
    print(11**i, palinChecker(str(11**i)))
```

2.4 Linked Lists (einfach verkettete Listen)

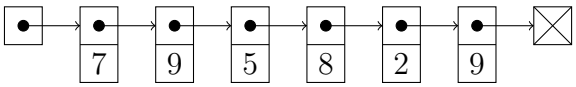


Abbildung 2.4: Modell einer Linked List

- Die einfach verkettete Liste wird durch den ersten Zeiger links repräsentiert.
- Dieser Zeiger zeigt auf den ersten Knoten, der wiederum aus einer Zeigevariablen (Punkt) und einem Datenfeld besteht.
- Der Zeiger eines Knotens zeigt entweder auf einen weiteren Knoten oder auf den Nullwert, der das Ende der Liste kennzeichnet (Quadrat mit Kreuz).

Die Klasse `Node`

Die Knoten werden als Objekte der Klasse `Node` implementiert.

Node
data: <any> next: Node
Node(data: <any>) getData(): <any> getNext(): Node setData(newData: <any>) getNext(newNext: Node)

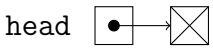
```
1  class Node:
2
3      def __init__(self, data):
4          self.data = data
5          self.next = None
6
7      def getData(self):
8          return self.data
9
10     def getNext(self):
11         return self.next
12
13     def setData(self, newData):
14         self.data = newData
15
16     def setNext(self, newNext):
17         self.next = newNext
```

Die Klasse `Linkedlist`

Linkedlist
head: Node
add(item: <any>) isEmpty(): bool length(): int search(item: <any>) bool remove(item: <any>)

Der Konstruktor

Es wird eine Instanzvariable `head` erzeugt der mit `None` initialisiert wird. Sobald später der erste Knoten hinzugefügt wird, ersetzt man Node durch die Adresse des Knotens.




```
1  from node import Node
2
3  class Linkedlist:
4
5      def __init__(self):
6          self.head = None
```

Die Methode `isEmpty()`

```
1      def isEmpty(self):
2          return self.head == None
```

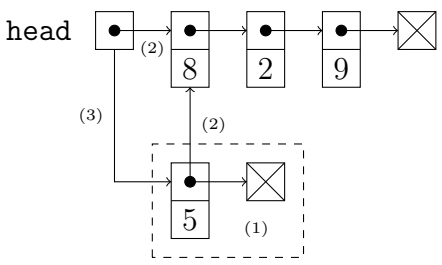
Die Methode `add(item)`

Möchte man einer einfach verketteten Liste das Element „5“ hinzufügen, so erfolgt dies effizient am Listenkopf.

(1) Erzeuge einen neuen „isolierten“ Knoten mit dem Wert 5.

(2) Kopiere den Wert der `next`-Variable des Listenkopfs in die `next`-Variable des neuen Knotens.

(3) Der Variable `next` des Listenkopfs wird die Adresse des neuen Knotens zugewiesen.



```
    def add(self, data):
        temp = Node(data)
        temp.setNext(self.head)
        self.head = temp
```

Die Methode `length()`

Man beginnt bei **head** und geht dann mit einer **while**-Schleife von Referenz zu Referenz, bis man mit **None** das Ende der Liste erreicht hat.

Erhöht man bei jeder Iteration einen Zähler um 1, so erhält man am Ende die Länge der Datenstruktur.

```
    def length(self):
        count = 0
        node = self.head
        while node != None:
            count += 1
            node = node.getNext()
        return count
```

Die Methode `search(item)`

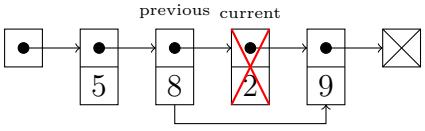
Diese Methode ist ähnlich wie **length** aufgebaut. Zusätzlich überprüft man bei jedem Schleifendurchlauf, ob sich **item** im Knoten befindet. Wenn ja, liefert die Methode **True** als Rückgabewert.

Erricht die Schleife das Ende der einfach verketteten Liste, ohne **item** gefunden zu haben, so liefert die Methode **False** zurück.

```
    def search(self, item):
        node = self.head
        while node != None:
            if node.getData() == item:
                return True
            node = node.getNext()
        return False
```

Die Methode `remove(item)`

Um den Knoten mit dem Datenwert **item** aus der Linkedlist zu entfernen muss man ihn wie in **search(item)** zuerst finden. Das Problem dabei ist, dass die Referenz auf den zu löschenden Knoten beim Erreichen des gesuchten Knotes bereits übersprungen wurde; also nicht mehr zur Verfügung steht, um auf den (allfälligen) übernächsten Knoten zu zeigen.



Die Lösung besteht darin, beim Durchlaufen der einfach verketteten Liste jeweils eine Referenz auf den aktuellen *und* den davor liegenden Knoten zu speichern.

Man beachte, dass der Listenkopf `head` keinen Vorgänger hat, weshalb man seinen Vorgängerknoten mit `None` initialisieren sollte.

Weiter gilt es zu beachten, dass das Entfernen des ersten Knotens getrennt von dem Entfernen eines „inneren“ Knotens behandelt werden muss.

Muss man auch das Entfernen des letzten Knotens separat behandeln?

```
1     def remove(self, item):
2         prevNode = None
3         currNode = self.head
4         while currNode != None:
5             if currNode.getData() == item:
6                 if prevNode == None:
7                     head = currNode.getNext()
8                 else:
9                     prevNode.setNext(currNode.getNext())
10
11            prevNode = currNode
12            currNode = currNode.getNext()
13        return False
```

Bemerkung

Das Entfernen eines Elements, hinterlässt einen verwaisten Knoten. In vielen Programmiersprachen muss der Programmierer dieses Objekt explizit mit einem sogenannten *Destruktor* löschen, wenn er nicht Speicherplatz verschwenden will.

Auch Python kennt eine `del()` Methode für Objekte. Diese wird in der Regel aber nicht benötigt, da ein spezieller Mechanismus, der *Garbage Collection* genannt wird, von Zeit zu Zeit dafür sorgt, dass nicht mehr verwendete Speicherbereiche freigegeben werden.

3 Bäume (Trees)

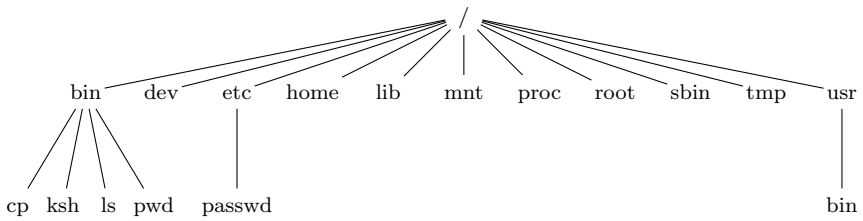
Überblick

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!

3.1 Beispiele

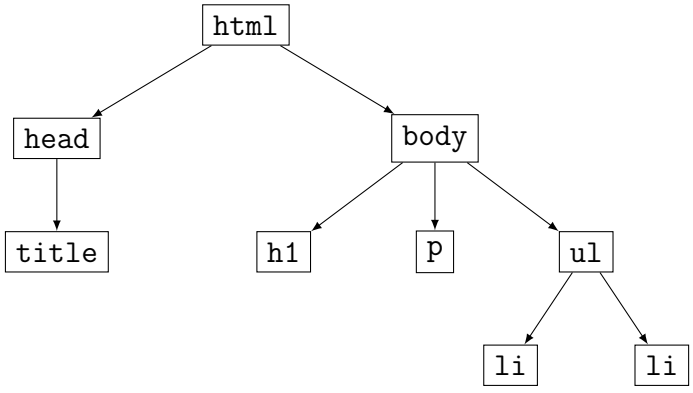
Unix-Dateisystem



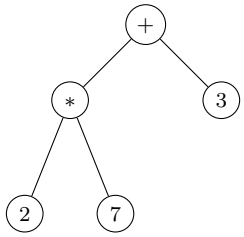
Andere Betriebssysteme haben ebenfalls eine mehr oder weniger feste Verzeichnisstruktur.

HTML-Dokumente

```
<html>
  <head>
    <title>Simple HTML-Document</title>
  </head>
  <body>
    <h1>Caption</h1>
    <p>Paragraph</p>
    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
    </ul>
  </body>
</html>
```



Syntaxbäume (Parse Trees)



3.2 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 Baums (height): Das Maximum der Menge aller Knotenlevels.

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

Wald (forest): Eine Menge von Bäumen.

Definition eines Baums (rekursiv)

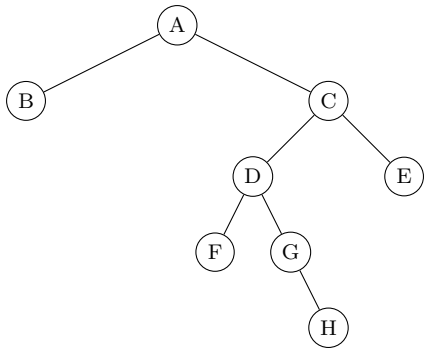
Ein Baum ist entweder leer oder besteht aus einer Wurzel und null oder mehr Teilbäumen, die jeweils selber ein Baum sind. Die Wurzel jedes Teilbaums ist mit der Wurzel des Elternbaums durch eine Kante verbunden.

Bemerkung

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

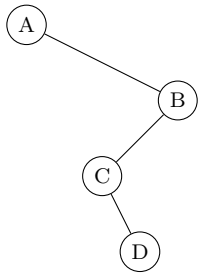
3.3 Typologie

Binärbaum (*binary tree*)



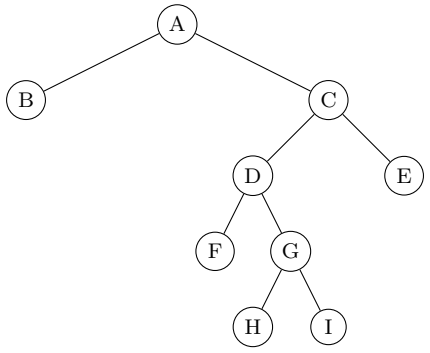
Ein Baum, bei dem jeder innere Knoten maximal zwei Blätter hat.

Entarteter Binärbaum (*degenerated binary tree*)



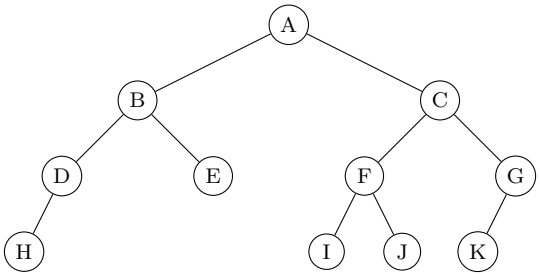
Bei einem entarteteten Baum hat jeder innere Knoten genau ein Kind. Also handelt es sich um einen „Unärbaum“; d. h. um eine lineare Datenstruktur.

Voller Binärbaum (*full binary tree*)



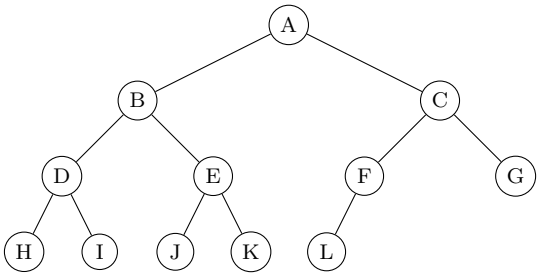
Jeder innere Knoten hat entweder null oder zwei Blätter. Das Adjektiv „voll“ bezieht sich darauf, dass alle inneren Knoten mit zwei Knoten oder Blättern „aufgefüllt“ sind.

Balancierter Binärbaum (*balanced binary tree*)



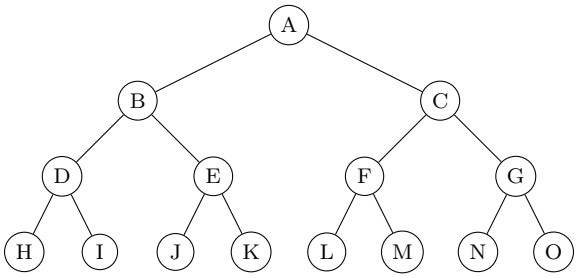
Ein balancierter Binärbaum ist ein Binärbaum, bei dem sich die Tiefe der Blätter höchstens um 1 unterscheiden.

Vollständiger Binärbaum (*complete binary tree*)



Ein vollständiger Binärbaum ist ein balancierter Binärbaum, bei dem alle Blätter mit der Tiefe n links von denjenigen mit der Tiefe $n-1$ stehen.

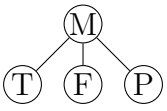
Perfekter Binärbaum (*perfect binary tree*)



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

n-äre Bäume

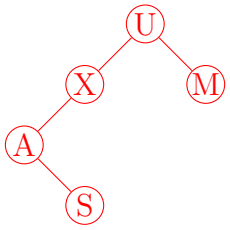
Entsprechende Baumstrukturen können auch für mehr als zwei Verzweigungen definiert werden. Im Fall von drei Verzweigungen spricht man von ternerären Bäumen (*ternary trees*); im Fall von n Verzweigungen spricht man von n-ären Bäumen (n-*ary trees*).



3.4 Darstellung binärer Bäume als verschachtelte Listen

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

Repräsentation als Baum:



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

3.5 Darstellung binärer Bäume durch Knoten und Referenzen

Die Datenstuktur wird analog zur einfach verketteten Liste definiert. Dabei gehen von der Wurzel eines (Sub)Baums jeweils zwei Referenzen aus. Diese Referenzen können ins Nichts (`None`) oder auf weitere (Sub)Bäume verweisen.

```
1 class BinaryTree:
2
3     def __init__(self, value):
4         self.value = rootObj
5         self.leftChild = None
6         self.rightChild = None
```

Speichertechnisch lässt sich dies so visualisieren:

Bei Adresse `07` (Zeile/Kolonne) beginnt eine Binärbaumstruktur aus drei aufeinanderfolgenden Zellen: `key`, `leftChild`, `rightChild`.

Die Speicherzelle `00` wird nicht verwendet, damit dieser Adresswert als leerer Zeiger (`NULL` oder `None`) verwendet werden kann. Grau hinterlegte Zellen sind besetzt.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				'Q'	00	00		'B'	2C	11						
1		'W'	00	03												
2													'C'	00	00	
3																

Knoten einfügen

```
    def insertLeft(self, value):
        if self.leftChild == None:
            self.leftChild = BinaryTree(value)
        else:
            tmp = BinaryTree(value)
            tmp.leftChild = self.leftChild
            self.leftChild = tmp

    def insertRight(self, value):
        if self.rightChild == None:
            self.rightChild = BinaryTree(value)
        else:
            tmp = BinaryTree(value)
            tmp.rightChild = self.rightChild
            self.rightChild = tmp
```

Knoten abfragen und verändern

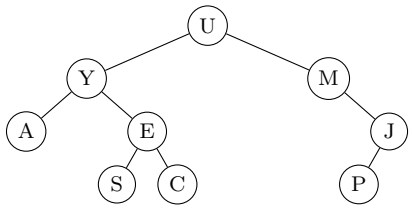
```
    def getRootVal(self):
        return self.value

    def setRootVal(self, value):
        self.value = value

    def getLeftChild(self):
        return self.leftChild

    def getRightChild(self):
        return self.rightChild
```

Maximale Tiefe eines Baumes bestimmen

```
    def maxDepth(self):
        if self.leftChild == None and self.rightChild == None:
            return 1
        elif self.leftChild == None:
            return 1 + self.rightChild.maxDepth()
        elif self.rightChild == None:
            return 1 + self.leftChild.maxDepth()
        else:
            return 1 + max(self.leftChild.maxDepth(),
                           self.rightChild.maxDepth())
```

Traversierung eines Baumes

Traversierung: systematisches Durchlaufen aller Knoten eines Baumes



Preorder

```
    def preorder(self):
        yield self.value
        if self.leftChild != None:
            yield from self.leftChild.preorder()
        if self.rightChild != None:
            yield from self.rightChild.preorder()
```

Inorder

```
    def inorder(self):
        if self.leftChild != None:
            yield from self.leftChild.inorder()
        yield self.value
        if self.rightChild != None:
            yield from self.rightChild.inorder()
```

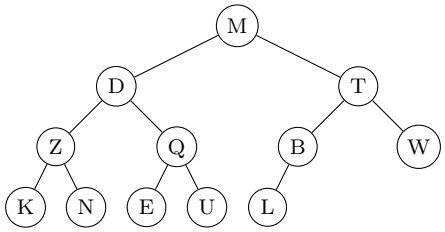
Postorder

```
    def postorder(self):
        if self.leftChild != None:
            yield from self.leftChild.postorder()
        if self.rightChild != None:
            yield from self.rightChild.postorder()
        yield self.value
```

3.6 Darstellung vollständiger Binärbäume als Heap

Heap: Darstellung eines vollständigen Binärbaums als Liste

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



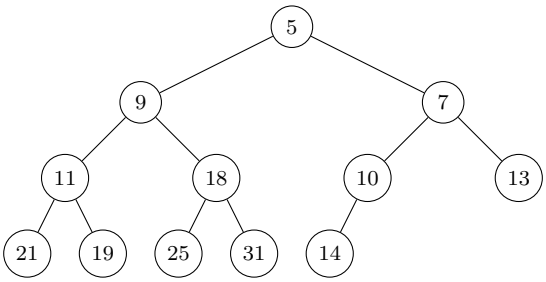
Elternknoten des Kindknotens mit Index i: $\lfloor i/2 \rfloor$

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

rechter Kindknoten des Elternknotens mit Index i: $2i + 1$

Min- und Max-Heaps

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

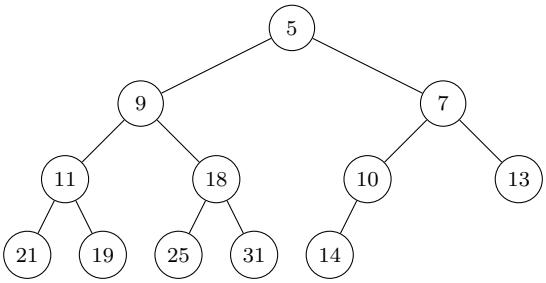


Min-Heap-Eigeschaft: Der kleinste Schlüssel ist in der Wurzel.

```
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[j],self.heap[i] \
12         = self.heap[i],self.heap[j]
```

Schlüssel hinzufügen

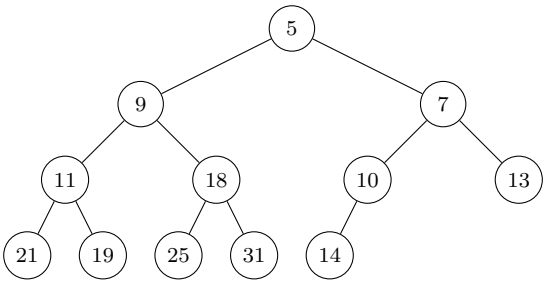
Ein neuer Schlüssel (3) wird an die letzte Position des Heaps gesetzt und so lange „hochgetrieben“ (`swim`), bis die Min-Heap-Eigenschaft wieder hergestellt ist.




```
    def insert(self, item):
        self.heap.append(item)
        self.size += 1
        self.swim(self.size)

    def swim(self, i):
        while i // 2 > 0:
            if self.heap[i] < self.heap[i//2]:
                self.swap(i, i//2)
            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.

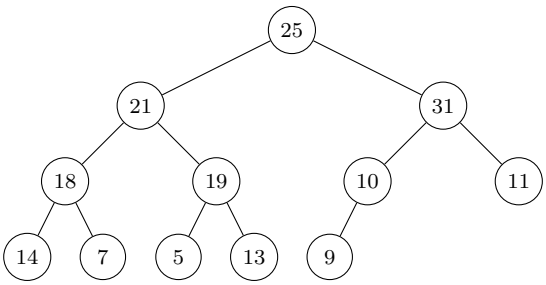



```
    def delMin(self):
        top = self.heap[1]
        self.heap[1] = self.heap.pop()
        self.size -= 1
        self.sink(1)
        return top

    def sink(self, i):
        while 2*i+1 <= self.size:
            j = self.minChild(i)
            if self.heap[i] > self.heap[j]:
                self.swap(i, j)
            i = j
```

```
    def minChild(self, i):
        # falls der letzte Elternknoten nur ein Kind hat:
        if 2*i+1 > self.size:
            return 2*i+1
        else:
            if self.heap[2*i] < self.heap[2*i+1]:
                return 2*i
            else:
                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.




```
1       def buildHeap(self, L):
2           self.heap = [None] + L[:]
3           self.size = len(L)
4           i = self.size//2
5           while i > 0:
6               self.sink(i)
7               i = i-1
```