

Algorithmen en Datastructuren.

Inhoudsopgave

1. Inleiding.....	4
2. De 'grote O' notatie.....	6
3. Zoeken.....	10
Sequential Search.....	10
Binary Search.....	12
4. Sorteren.....	14
Insertion sort.....	16
5. Datastructuren.....	20
Set (verzameling).....	20
De afbeelding (dictionary, map).....	22
Stack.....	24
Queue.....	25
6. Recursie.....	27
Wat is recursie?.....	27
Hoe werkt recursie?.....	30
De getallen van Fibonacci.....	31
Alle permutaties van een lijst tonen.....	32
Mergesort.....	33
Quicksort.....	35
Backtracking.....	36
7. Recursieve datastructuren.....	39
Lijst-structuren.....	40
8. Boom-structuren.....	42
Binaire bomen.....	44
Een boom opbouwen.....	48
Een boom afdrukken.....	49
Een boom doorlopen.....	50
9. Binary Search Trees.....	51
Zoeken in een binaire boom.....	52
Een element toevoegen.....	53
Een element verwijderen.....	54
10. Gebalanceerde bomen.....	55
De AVL-tree.....	56
Red black tree.....	59
11. Hashing.....	60
Linear hashing.....	62
Quadratic hashing.....	66
Separate chaining hashing.....	68

Hashfuncties.....	69
12. Dynamisch programmeren.....	71
13. Grafen.....	78
Inleiding.....	78
Terminologie.....	78
Wiskundige notaties.....	80
Representaties van een graaf.....	81
Een implementatie in Python.....	81
Het 'breadth first search' (BFS) algoritme.....	83
De implementatie van BFS.....	85
Het 'depth first search' (DFS) algoritme.....	88
De implementatie van DFS.....	90
De minimale opspanningsboom.....	92
Het MST-algoritme van Kruskal.....	92
De implementatie van het Kruskal-algoritme.....	95
Het MST-algoritme van Prim.....	96
De implementatie van het Prim-algoritme.....	99
Het 'korste pad' bepalen.....	100
Het algoritme van Dijkstra.....	100
De implementatie van het Dijkstra-algoritme.....	102
De performance van algoritmen.....	103

1. Inleiding.

Het doel van het vak is:

- goed leren programmeren
- leren denken in algoritmen
- leren werken met datastructuren
- het begrip 'recursiviteit' eigen maken

In de informatica komen veel interessante algoritmische problemen voor.

Een klein voorbeeld: bepaal van drie gegeven getallen het middelste element.

Een groter voorbeeld: herken in een afbeelding het kenteken van een auto.

Op de volgende vragen wordt kort ingegaan:

- wanneer is een programma efficiënt?
- wat zijn datastructuren?
- wat is een algoritme?
- wat wordt in dit vak behandeld?
- welke programmeertaal gebruiken we?

Wanneer is een programma efficiënt?

We onderscheiden twee criteria:

- Het moet snel zijn
- Het heeft weinig geheugenruimte nodig

De kunst is het zoeken van een goed evenwicht tussen beide criteria.

Wat zijn datastructuren?

M.b.v. datastructuren worden gegevens geordend.

Voorbeelden van datastructuren zijn:

- lijsten, tabellen, arrays
- boomstructuren
- netwerken (ook wel grafen genoemd)

Wat is een algoritme?

Letterlijk betekent algoritme : rekenvoorschrift.

Een algoritme is een rij opdrachten, waarmee een probleem opgelost kan worden.

Voorbeelden van problemen zijn: zoeken, sorteren, het grootste element bepalen, de kortste verbinding bepalen, etc.

Wat wordt in dit vak behandeld?

Een aantal basialgoritmen en datastructuren. Deze worden bij het programmeren vaak gebruikt.

Welke programmeertaal gebruiken we?

Dit vak is onafhankelijk van een programmeertaal.

We kiezen voor de programmeertaal Python, omdat deze taal het beste aansluit bij het vak.

De afstand tussen pseudo-code en python-code is klein.

2. De 'grote O' notatie.

Om na te gaan hoe efficiënt een algoritme is, wordt uitgezocht, hoeveel deelstappen nodig zijn.

Dit wordt vastgelegd m.b.v. een formule.

Het aantal deelstappen wordt aangegeven met 'n'.

Voorbeelden van dergelijke formules zijn:

$$3n + 4$$

$$10n^3 + n^2 + 40n + 80$$

$$2^n + n^2$$

$$n^2 \log(n) + 2n + 8$$

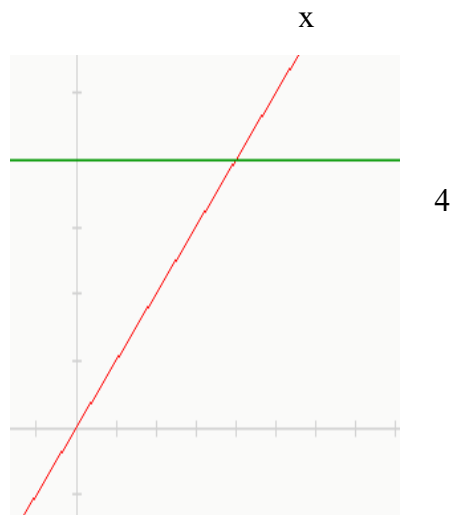
Bij dergelijke functies domineert één term als 'n' steeds groter wordt. De constante voor die term is niet belangrijk en wordt weggelaten.

De dominante term wordt met een grote O aangegeven.

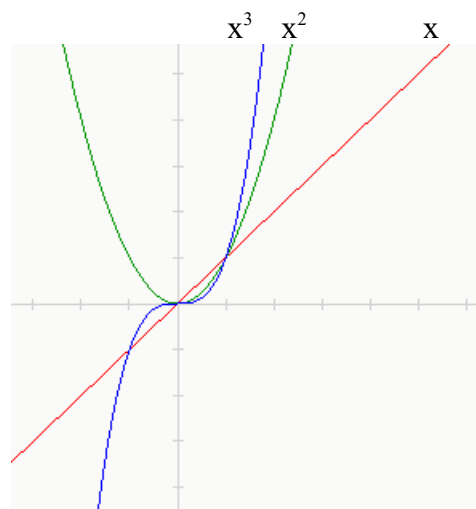
Voorbeeld 1. $3n + 4 = O(n)$

Toelichting:

- n stijgt
- 4 blijft constant
- de constante '3' voor '3n' wordt weggelaten.

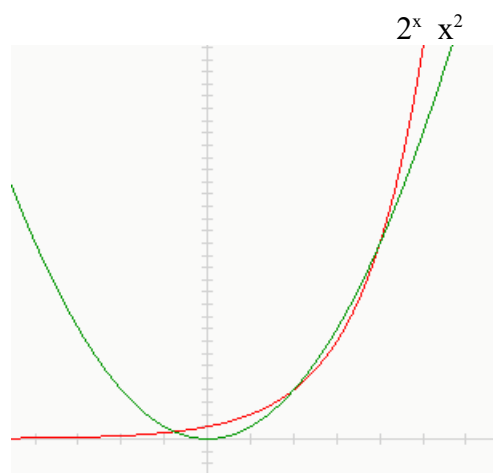


Voorbeeld 2. $10n^3 + n^2 + 40n + 80 = O(n^3)$



n	1	5	10	50	100	500	1000
n^2	1	25	100	2500	10000	250000	1000000
n^3	1	125	1000	125000	1000000	125000000	1000000000

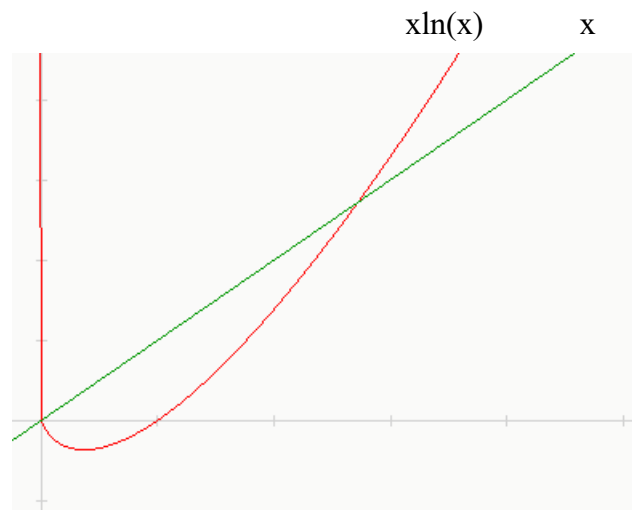
Voorbeeld 3. $2^n + n^2 = O(2^n)$



n	1	5	10	50	100	500	1000
n^2	1	25	100	2500	10000	250000	1000000
2^n	2	32	1024	$1.13 \cdot 10^{15}$	$1.27 \cdot 10^{30}$	$3.27 \cdot 10^{150}$	$1.07 \cdot 10^{301}$

Voorbeeld 4

$$n^2 \log(n) + 2n + 8 = O(n \log(n))$$

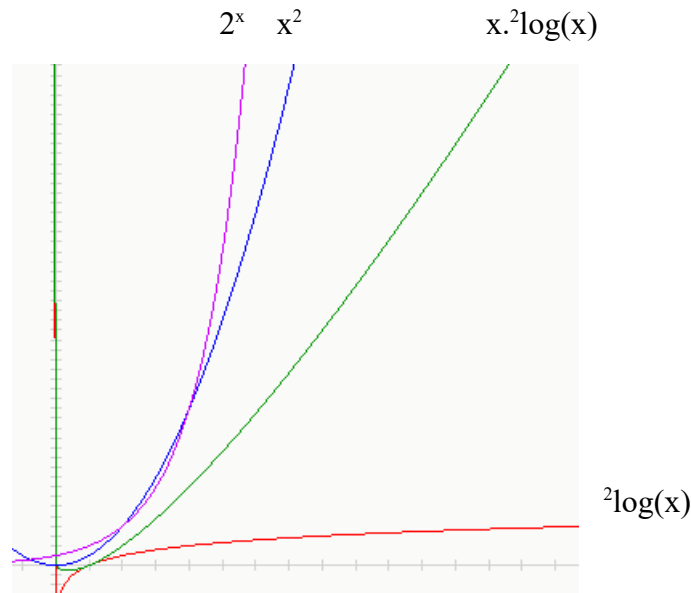


n	1	5	10	50	100	500	1000
$n \cdot \log n$	0	11.6	33.2	282.2	664.4	4482.9	9965.8

logaritmische functies, zoals $^2\log(n)$, $^3\log(n)$, etc, nemen niet snel toe voor grote n

polynomiale functies, zoals n^2 , n^3 , etc, groeien sneller

exponentiële functies, zoals 2^n , 3^n , etc groeien zeer snel.



n	1	100	1000	10⁶	10⁹
$^2\log(n)$	0	6.6	10	19.9	29.9
$n.^2\log(n)$	0	664	9966	$2.0 * 10^7$	$3.0 * 10^{10}$
n^2	1	10^4	10^6	10^{12}	10^{18}
2^n	2	$1.3 * 10^{30}$	$1.1 * 10^{301}$	-	-

Er wordt nogal gemakkelijk gedaan over grote getallen.

De computers worden steeds sneller en de geheugens steeds groter.

Om een indruk te krijgen hoe groot getallen werkelijk zijn geef ik enkele voorbeelden.

Omtrek aarde: 40.000 km = 4.10^7 m

Lichtsnelheid: 300.000 km/s = $3 * 10^8$ m/s

Afstand tussen aarde en maan: 384.450 kilometer = $3.8 * 10^8$ m

Afstand tussen aarde en zon: 149,6 miljoen kilometer = $1.5 * 10^{11}$ m

Levensduur 100-jarige: $100 * 365 * 24 * 60 * 60$ sec = 3153600000 sec = $3.1 * 10^9$ sec

1 gigabyte = 2^{30} byte = 1073741824 byte = $1.1 * 10^9$ bytes

1 nanoseconde = 10^{-9} seconde = 1 miljardste seconde

1 gigahertz = 1 GHz = 10^9 hertz

3. Zoeken.

Ga uit van het volgende probleem:

Gegeven een integer x en een rij a.

Geef de positie van x in a of geef aan dat x niet in a voorkomt.

Als x meer dan één keer voorkomt geef dan de kleinste positie.

Voorbeeld 1:

	0	1	2	3	4	5
a :	45	65	34	82	30	22
x :	30					

Vraag: Geef de positie van x.

Antwoord: 4

Voorbeeld 2:

	0	1	2	3	4	5
a :	45	65	15	82	15	22
x :	15					

Vraag: Geef de positie van x.

Antwoord: 2

Sequential Search.

Deze wordt toegepast als de rij **niet** gesorteerd is.

- x is niet aanwezig: alle n elementen moeten geraadpleegd worden
- x is aanwezig

Het ergste geval (worst case): x is het laatste element

Alle n elementen moeten worden geraadpleegd

Het beste geval (best case): x is het eerste element.

Alleen het eerste element moet worden geraadpleegd.

Gemiddeld: $\frac{n+1}{2} = \frac{1}{2}n + \frac{1}{2}$ elementen moeten worden geraadpleegd.

Sequential search is een $O(n)$ –algoritme.

Implementaties van sequential search:

1. Python heeft bij een lijst a een ingebouwde methode, die bij een gegeven x de positie van x bepaalt:

a.index(x)

2. Als we geen gebruik willen maken van de index-methode, kan het als volgt worden opgelost:

```
def sequentialSearch1(a,x): # a is een lijst,
                           # x is het gezochte element
    for i in range(len(a):
        if a[i] == x:
            return i
    return -1
```

3. De vorige oplossing is een oplossing zoals in de taal C wordt gebruikt. In Python kunnen de elementen van een lijst ook geraadpleegd worden zonder een index te gebruiken:

```
def sequentialSearch2(a,x):
    i = 0
    for e in a:
        if e == x:
            return i
        else:
            i += 1
    return -1
```

4. Python kent de ingebouwde functie 'enumerate' . Op basis van een lijst kan hiermee een lijst met (index, element)-combinaties worden gemaakt.

Voorbeeld:

```
a = [45,65,34,82,30,22]
enum = enumerate(a)
print(list(enum))
```

Uitvoer:

```
[(0, 45), (1, 65), (2, 34), (3, 82), (4, 30), (5, 22)]
```

Bij de volgende oplossing wordt van 'enumerate' gebruik gemaakt.

```
def sequentialSearch3(a,x):
    enum = enumerate(a)
    for i,v in enum:
        if v == x:
            return i
    return -1
```

Binary Search.

'Binary search' wordt toegepast als de rij gesorteerd is.

Wat houdt "gesorteerd" in?

Sorteren is alleen mogelijk als de elementen onderling vergelijkbaar zijn.

Wanneer zijn twee elementen onderling vergelijkbaar?

Je moet bij 2 elementen kunnen aangeven of de één groter dan, gelijk aan of kleiner dan de ander is.

Getallen zijn onderling te vergelijken. Hetzelfde geldt voor strings.

Bij andere objecten moet je aangeven hoe je vergelijkt. Bijvoorbeeld bij boeken: het aantal bladzijden of de hoogte of de naam van de auteur of het ISBN-nummer.

Bij Python worden voor het vergelijken de vergelijkingsoperatoren gebruikt.

Een vergelijkingsoperator correspondeert met een methode:

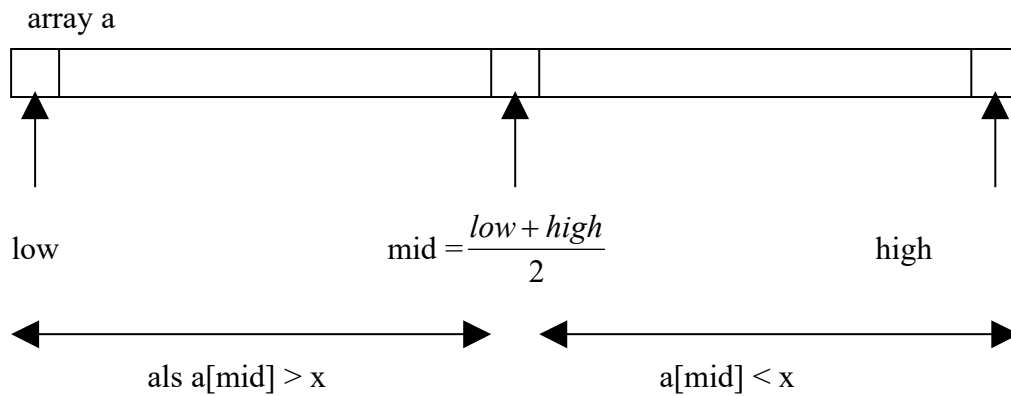
$a == b$	<code>a.__eq__(b)</code>
$a != b$	<code>a.__ne__(b)</code>
$a < b$	<code>a.__lt__(b)</code>
$a \leq b$	<code>a.__le__(b)</code>
$a > b$	<code>a.__gt__(b)</code>
$a \geq b$	<code>a.__ge__(b)</code>

De methoden worden "rich comparison methods" genoemd.

Als je de ingebouwde sort-functie van Python gebruikt, hoef je alleen de methode `__lt__` te implementeren.

Binary Search werkt als volgt:

- Vergelijk het middelste element van de rij met x
- Als het middelste element groter is dan x zoek dan verder in linkerhelft van de rij
- Als het middelste element kleiner is dan x zoek dan verder in rechterhelft van de rij
- Als het middelste element gelijk is aan x dan is de positie gevonden



Implementatie van Code 'binary search':

```
def binarySearch(a,x): # a is een lijst,
                        # x is het gezochte element
    low = 0
    high = len(a) - 1
    while low <= high:
        mid = ( low + high ) // 2 # gehele deling
        if a[mid] < x:
            low = mid + 1
        elif a[mid] > x:
            high = mid - 1
        else:
            return mid
    return -1
```

Het volgende geldt:

Bij 'Linear Search' is het aantal stappen: $O(n)$

Bij 'Binary Search' is het aantal stappen: $O(\log(n))$. Dit is een hele verbetering

De volgende tabel laat zien, dat het om een grote verbetering gaat.

n	1	10	100	1000	10^6	10^9
${}^2\log n$	0	3.3	6.6	10.0	19.9	29.9

Om 'Binary Search' toe te kunnen passen moet de rij gesorteerd zijn.

4. Sorteren.

Ga uit van een lijst a:

```
a = [45, 65, 34, 82, 30, 22]
```

Een lijst wordt als volgt gesorteerd:

```
a.sort()
```

De lijst 'a' is nu gewijzigd. Dit is mogelijk omdat de klasse 'list' een mutable klasse is.

Het is ook mogelijk een gesorteerde versie van 'a' te maken zonder 'a' te wijzigen. Dit gaat m.b.v. de functie 'sorted' .

Een voorbeeld:

```
a = [45, 65, 34, 82, 30, 22]
b = sorted(a)

print('origineel:', a)
print('gesorteerd:', b)
```

Uitvoer:

```
origineel: [45, 65, 34, 82, 30, 22]
gesorteerd: [22, 30, 34, 45, 65, 82]
```

Als de lijst van groot naar klein gesorteerd moet worden, gaat dit als volgt:

```
a.sort(reverse=True)
```

Het sorteren kan ook op basis van een key-functie:

```
def neg(x):
    return -x

a.sort(key=neg)
```

De key-functie kan anoniem zijn:

```
a.sort(key=lambda x: -x)
```

Hierbij is "`lambda x: -x`" een functie (zonder naam) die aan iedere x de waarde -x toekent.

De term "lambda" is afkomstig van het wiskunde-onderwerp lambda-calculus. (zie <http://nl.wikipedia.org/wiki/Lambdacalculus>)

Een voorbeeld met strings:

```
a = ['100', '3000', '21', '20']

print('a:', a)
print('sorted(a):', sorted(a))
print('sorted(a,key=str):', sorted(a,key=str))
print('sorted(a,key=int):', sorted(a,key=int))
print('sorted(a,key=len):', sorted(a,key=len))
```

Uitvoer:

```
a: ['100', '3000', '21', '20']
sorted(a): ['100', '20', '21', '3000']
sorted(a,key=str): ['100', '20', '21', '3000']
sorted(a,key=int): ['20', '21', '100', '3000']
sorted(a,key=len): ['21', '20', '100', '3000']
```

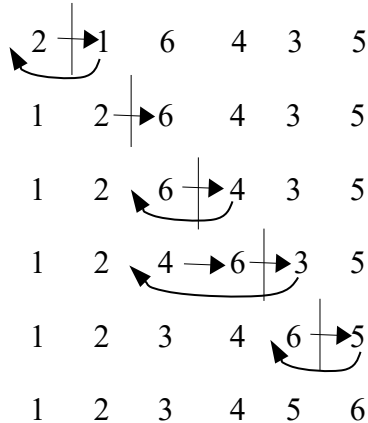
Insertion sort.

Een eenvoudige sorteer-methode is 'insertion sort'.

De methode is handig voor kleine rijen of bijna gesorteerde rijen.

Ga uit van de rij: 2 1 6 4 3 5

De rij wordt van links naar rechts gesorteerd



Na elke stap is de verticale streep 1 plaats verschoven en is de deelrij links van de streep gesorteerd. De elementen 1, 4, 3 en 5 moeten op de juiste positie worden tussengevoegd. Een aantal elementen moeten 1 plaats naar rechts worden verschoven.

Een implementatie:

```
def insertionSort(a):  
    for p in range(1, len(a)):  
        tmp = a[p]  
        j = p  
        while j > 0 and tmp < a[j-1]:  
            a[j] = a[j-1]  
            j -= 1  
        a[j] = tmp
```


Insertion sort met een key-functie:

```
def insertionSort2(a, key):  
    for p in range(1, len(a)):  
        tmp = a[p]  
        j = p  
        while j > 0 and key(tmp) < key(a[j-1]):  
            a[j] = a[j-1]  
            j -= 1  
        a[j] = tmp
```

Voor Insertion Sort geldt:

Aantal keyvergelijkingen: $O(n^2)$
Aantal verplaatsingen: $O(n^2)$

Een snelle methode is quicksort. Deze methode is $O(n \cdot \log(n))$. We gaan hier nu niet dieper op in. De sort-methoden van de klasse `java.util.Arrays` maken gebruik van quicksort.

Samenvatting:

Linear Search : $O(n)$
Binary Search : $O(\log(n))$
Insertion Sort : $O(n^2)$
Quicksort : $O(n \cdot \log(n))$

Wanneer moet je "linear search" toepassen en wanneer "binary search"?

Als je één keer moet zoeken en de rij is ongesorteerd, dan kun je beter niet eerst sorteren.

Als je k keer moet zoeken, dan is het zinvol eerst te sorteren als

$$k * n > n * \log(n) + k * \log(n)$$

We werken de uitdrukking uit:

$$\begin{aligned} k * (n - \log(n)) &> n * \log(n) \\ k &> n * \log(n) / (n - \log(n)) \approx \log(n) \end{aligned}$$

Conclusie: Als meer dan $\log(n)$ keer gezocht moet worden dan kun je beter eerst sorteren.

De snelheid van een algoritme kunnen we nagaan op de volgende manieren:

1. M.b.v. een wiskundige redenering
2. Door de tijdsduur te bepalen.

Voorbeeld:

```
import timeit
timer = timeit.default_timer

import random

a = [0]*1000
for i in range(1000):
    a[i] = random.randint(0,100000)

t1 = timer()
print('t1:', t1)

insertionSort(a)

t2 = timer()
print('t2:', t2)

print('t2-t1:', t2-t1)
```

Uitvoer:

```
t1: 3.0186054767162884e-07
t2: 0.07333279400915482
t2-t1: 0.07333249214860715
```

3. Door m.b.v. tellers na te gaan hoe vaak een bepaalde operatie wordt uitgevoerd

```
v = 0
```

```
def insertionSort3(a):  
    global v  
    for p in range(1, len(a)):  
        tmp = a[p] ; v += 1  
        j = p  
        while j > 0 and tmp < a[j-1]:  
            a[j] = a[j-1] ; v += 1  
            j -= 1  
        a[j] = tmp ; v += 1
```

5. Datastructuren.

In dit hoofdstuk worden enkele datastructuren behandeld.

Datastructuren worden gebruikt om een aantal bewerkingen uit te voeren op gegevens:

- toevoegen (store)
- raadplegen (access, retrieve)
- wijzigen
- verwijderen (delete)

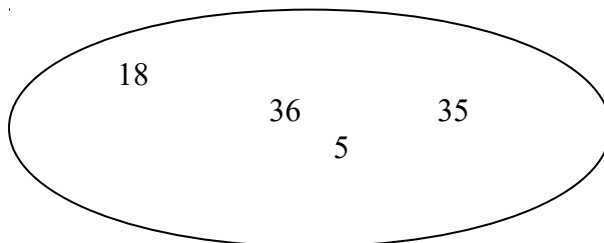
Behandeld zijn

- list
- dictionary

In dit hoofdstuk komen aan bod:

- set
- stack
- queue

Set (verzameling).



Bij een set worden de elementen niet achter elkaar geplaatst. Er is geen sprake van een index of positie van een element. Verder bevat een verzameling geen twee dezelfde elementen.

Python kent het type 'set' .

Voorbeeld 1:

```
s = {3, 24, 105}
print(s)
```

Uitvoer:

```
{24, 105, 3}
```

De volgorde is veranderd.

Het volgende is niet toegestaan:

```
s = {[3], [24], [105]}
```

De foutmelding `TypeError: unhashable type: 'list'` wordt dan getoond.

Sets maken gebruik van hashtechnieken. Dit heeft invloed op de positie van elementen in de lijst. Niet alle elementen lenen zich voor 'hashing'. Alleen objecten van klassen met de methode `__hash__()` zijn geschikt.

Iets dergelijks geldt ook voor dictionaries.

```
d = {3: 'drie', 24: 'vier_en_twintig', 105: 'honderd_en_vijf'}
print(d)
```

Uitvoer:

```
{24: 'vier_en_twintig', 105: 'honderd_en_vijf', 3: 'drie'}
```

Ook is niet toegestaan:

```
d = {[3]: 'drie', [24]: 'vier_en_twintig', [105]: 'honderd_en_vijf'}
```

Voorbeeld 2:

```
a = [3, 24, 105, 105, 3, 24]
s = set(a)
print('s:', s)
```

Uitvoer:

```
{24, 105, 3}
```

De elementen komen maar één keer voor.

Voorbeeld 3:

```
s2 = 'abacadabra'
s = set(s2)
print('s:', s)
print('sorted(s):', sorted(s))
```

Uitvoer:

```
s: {'b', 'd', 'a', 'r', 'c'}
sorted(s): ['a', 'b', 'c', 'd', 'r']
```

Merk op dat `'sorted(s)'` is van het type `'list'`.

(zie ook <http://effbot.org/zone/python-hash.htm>)

De afbeelding (dictionary, map).

Vaak worden gegevens bewaard en geraadpleegd m.b.v. een key.

Dit wordt gerealiseerd door (key, value) – combinatie op te slaan.

Een groep van (key, value) –combinaties wordt een afbeelding (map) of functie genoemd.

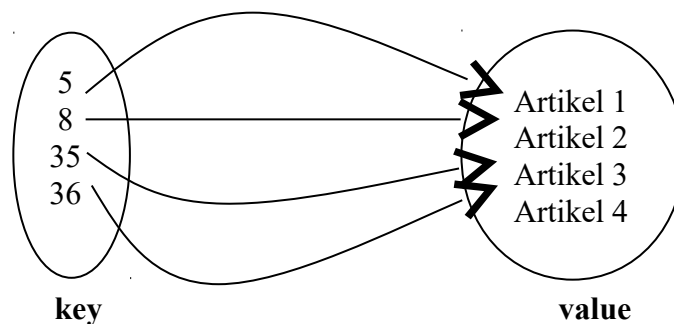
Een key komt precies één keer voor.

Voorbeeld :

	Artikelnr	naam	prijs
Artikel 1	5	tafel	180
Artikel 2	18	bank	330
Artikel 3	35	stoel	110
Artikel 4	36	kast	450

Artikelnr is een unieke key.

De volgende afbeelding kan gevormd worden: (5, artikel 1), (8, artikel 2), (35, artikel 3) en (36 , artikel 4)



In Python is dit als volgt te realiseren:

```
class Artikel:
    def __init__(self, nr, naam, prijs):
        self.nr = nr
        self.naam = naam
        self.prijs = prijs

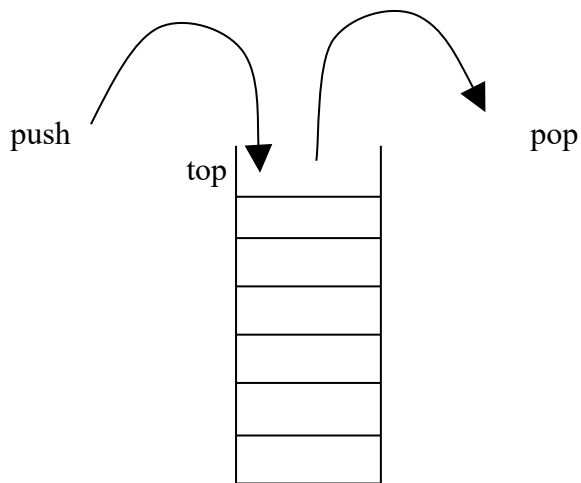
    def __repr__(self):
        return "<< nr: " + str(self.nr) + " naam: " + self.naam +
            " prijs: " + str(self.prijs) + ">>"

a =[ Artikel(35, 'stoel', 110),
      Artikel(18, 'bank', 330),
      Artikel(36, 'kast', 450),
      Artikel(5, 'tafel', 180)
    ]

d = {a[i].nr:a[i] for i in range(4)}
```

Stack

Een Stack is gebaseerd op het LIFO-principe: Last In First Out



In Python is een stack te realiseren met de klasse 'list'.

```
push: a.append(x)
pop:  a.pop()
```

Voorbeeld:

```
a = [2,6,3]
print('a:', a)

a.append(4)
print('a:', a)

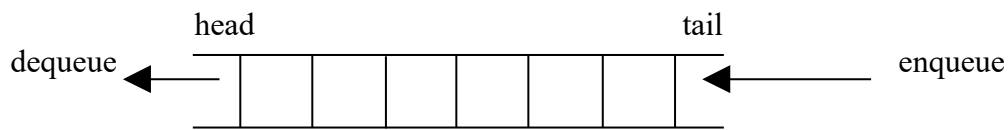
print('a.pop():', a.pop())
print('a:', a)
```

Uitvoer:

```
a: [2, 6, 3]
a: [2, 6, 3, 4]
a.pop(): 4
a: [2, 6, 3]
```


Queue

Een Queue is gebaseerd op het FIFO-principe : First In First Out



Enqueue: voeg een element achteraan toe.

Dequeue: haal een element vooraan weg

Queues worden binnen de informatica zeer veel gebruikt: eventqueue, printqueue, taskqueue, etc.

Python heeft een aantal queue-implementaties:

- **`collections.deque([iterable[, maxlen]])`**
zie <https://docs.python.org/3.6/tutorial/datastructures.html> en
<https://docs.python.org/3.6/library/collections.html#collections.deque>

Deque: double ended queue: aan beide zijden kunnen elementen toegevoegd en verwijderd worden.

- De module 'queue' bevat de thread-safe klassen Queue, LifoQueue en PriorityQueue
zie <https://docs.python.org/3/library/queue.html#module-queue>

We maken zelf een queue-klasse:

```
class myqueue(list):
    def __init__(self,a=[]):
        list.__init__(self,a)

    def dequeue(self):
        return self.pop(0)

    def enqueue(self,x):
        self.append(x)

q = myqueue('345')
print('q:', q)

q.enqueue(100)
print('q:', q)

print('q.dequeue():', q.dequeue())
print('q:', q)
```

Uitvoer:

```
q: ['3', '4', '5']
q: ['3', '4', '5', 100]
q.dequeue(): 3
q: ['4', '5', 100]
```

6. Recursie.

Wat is recursie?

Recursieve constructies bevatten een verwijzing naar zichzelf.

In de praktijk komen recursieve structuren ook voor: ieder mens heeft een vader. Iedere vader heeft een vader, etc.

Een technisch voorbeeld: iedere directory bestaat uit files en subdirectories. Iedere subdirectory bestaat uit files en subdirectories, etc.

Functionele programmeertalen zijn gebaseerd op recursie.

Wat is het voordeel van recursie?

- het is een andere benadering van problemen
- het sluit meer aan bij wiskunde
- het is eenvoudiger na te gaan of een algoritme correct is
- de code is korter en helderder
- problemen kunnen m.b.v. recursie opgesplitst worden in deel-problemen.

Wat is het nadeel van recursie?

- het vraagt een andere (abstracte) manier van denken
- recursieve code is (iets) trager dan niet-recursieve code

Recursieve code kan altijd omgezet worden naar niet-recursieve code. Soms moet in de niet-recursieve code dan een stack gebruikt worden.

Een risico bij een recursieve methode is, dat de methode zichzelf eindeloos kan aanroepen. Dit leidt dan tot een stack-overflow.

Voorbeeld: bereken $1 + 2 + 3 + \dots + n$ voor een gegeven n .

Oplossing 1. Zonder recursie.

```
def som1(n):  
    som = 0  
    for i in range(1,n+1):  
        som = som + i  
    return som
```

Oplossing 2. Maak gebruik van wiskunde:

$$1 + 2 + 3 + \dots + n = \frac{n * (n + 1)}{2}$$

```
def som2(n):  
    return (n*(n+1))//2
```

Oplossing 3. Los het probleem recursief op

Als $S(n) = 1 + 2 + \dots + n$

dan is $S(1) = 1$

en $S(n) = 1 + 2 + \dots + (n-1) + n = S(n-1) + n$ als $n > 1$

$S(n-1)$

```
def som3(n):  
    if n == 1:  
        return 1  
    else:  
        return som3(n-1) + n
```

'som3' roept zichzelf n keer aan.

We zijn er van uitgegaan dat $n \geq 1$ is.

We kunnen deze aanname als volgt in de code verwerken:

```
assert n >= 1
```

of

```
if n <= 0:  
    return 0
```

of

```
if n <= 0:  
    raise Exception('argumenten kleiner dan 1 zijn niet toegestaan')
```

Voorbeeld:

n-faculteit = $n! = n \cdot (n-1) \cdot \dots \cdot 2 \cdot 1$ is een belangrijk wiskundig begrip.

Het aantal rangschikkingen van n elementen is $n!$.

Het aantal groepen van k elementen uit een groep van n elementen is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Bereken $n!$

Oplossing 1 : zonder recursie.

```
def fac1(n):  
    fac = 1  
    for i in range(2,n+1):  
        fac *= i  
    return fac
```

Oplossing 2 : met recursie:

Als $\text{fac}(n) = n \cdot (n-1) \cdot \dots \cdot 2 \cdot 1$

dan is $\text{fac}(0) = 1$

en $\text{fac}(n) = n \cdot \text{fac}(n-1)$ ($n > 1$)

We spreken af dat $\text{fac}(0) = 1$

```
def fac2(n):  
    if n < 0:          # error  
        return -1  
    elif n == 0:      # basisgeval  
        return 1  
    else:  
        return n * fac2(n-1)
```

de methode fac2 roept zichzelf n keer aan.

Voor recursie gelden de volgende regels:

- er moet een basisgeval zijn, waarbij geen recursie wordt gebruikt.
- de methode moet eindigen: na een aantal recursieve aanroepen wordt een basisgeval bereikt.

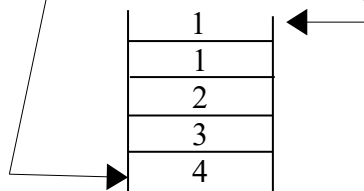
Hoe werkt recursie?

We illustreren dit aan de hand van een voorbeeld.
Neem als voorbeeld de methode fac2

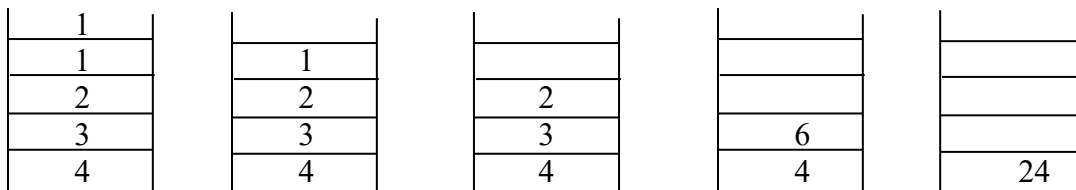
Volgens het algoritme is

$$\begin{aligned}\text{fac}(4) &= \\ 4 * \text{fac}(3) &= \\ 4 * (3 * \text{fac}(2)) &= \\ 4 * (3 * (2 * \text{fac}(1))) &= \\ 4 * (3 * (2 * (1 * \text{fac}(0)))) &= \\ 4 * (3 * (2 * (1 * 1))) &= \\ 4 * (3 * (2 * 1)) &= \\ 4 * (3 * 2) &= \\ 4 * 6 &= \\ 24\end{aligned}$$

De berekening wordt gemaakt door de waarden 4, 3, 2, 1 en 1 achtereenvolgens op een stack te plaatsen.



Vervolgens worden steeds de bovenste twee elementen verwijderd en het product op de stack gezet.



Resultaat: 24.

De juiste attitude bij het oplossen van recursieve problemen:

Bij het oplossen van een recursief probleem moet je uitgaan van een wiskundige relatie.
Je kunt je beter niet afvragen hoe het precies wordt uitgevoerd.

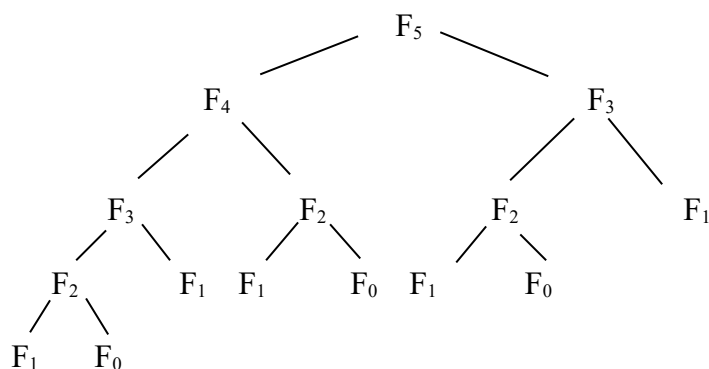
Deze zijn als volgt gedefinieerd:

$$F_n = F_{n-1} + F_{n-2} \text{ als } n > 1$$

Daarmee worden de volgende getallen verkregen: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

```
def fib1(n):  
    if n <= 1: # basisgevallen  
        return n  
    else:  
        return fib1(n-1) + fib1(n-2)
```

Dit is te zien in het volgende schema:



In dit geval is het beter geen gebruik te maken van recursie.

```
def fib2(n):
    if n <= 1:
        return n
    else:
        fvorig,f = 0,1
        for _ in range(2,n+1):
            fvorig,f = f, f+fvorig
        return f
```

Alle permutaties van een lijst tonen.

Een permutatie is een rangschikking.
Alle permutaties van [1,2,3] zijn:

```
[1,2,3]
[1,3,2]
[2,1,3]
[2,3,1]
[3,1,2]
[3,2,1]
```

Het volgende algoritme bepaalt alle permutaties van a

```
def perm(a):
    assert a
    if len(a) == 1:
        return [a]
    else:
        ap = []
        for i in range(len(a)): # bepaal alle permutaties die
                                # beginnen met a[i]
            ap2 = perm(a[:i]+a[i+1:]) # bepaal permutaties van lijst
                                      # zonder a[i]
            ap += [[a[i]]+p for p in ap2] # plaats voor iedere
                                          # permutatie a[i]
        return ap
```

In Python is de klasse 'permutations' aanwezig:

```
import itertools

p = itertools.permutations([7, 3, 9])
print(list(p))
```

Uitvoer:

```
[(7, 3, 9), (7, 9, 3), (3, 7, 9), (3, 9, 7), (9, 7, 3), (9, 3, 7)]
```

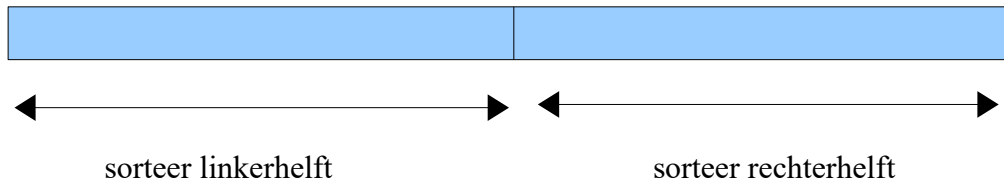

Mergesort.

Insertion sort : $O(n^2)$

Mergesort: $O(n \log(n))$, een snelle sorteermethode

Mergesort gaat als volgt:

- Splits de rij in twee (bijna) gelijke delen: een linker- en een rechterhelft
- Sorteer beide helften afzonderlijk
- Meng beide gesorteerde helften tot één geheel



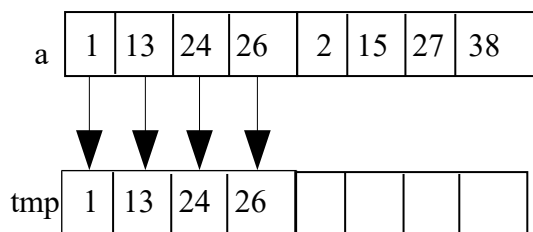
Het sorteren van beide helften gebeurt weer m.b.v. mergesort. De methode 'mergesort' is dus recursief.

Het mengen van twee deelrijen tot één geheel gebeurt m.b.v. een tijdelijke rij. Na afloop wordt het resultaat gekopieerd naar de oorspronkelijke rij.

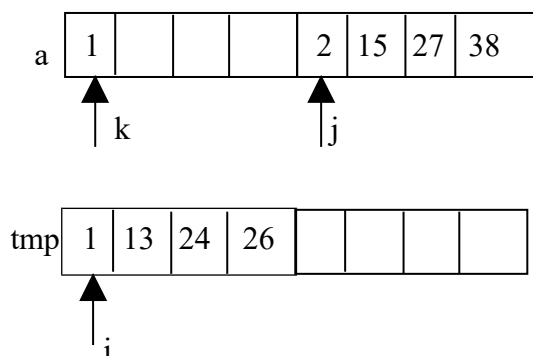
Het mengen van twee deelrijen gaat als volgt:

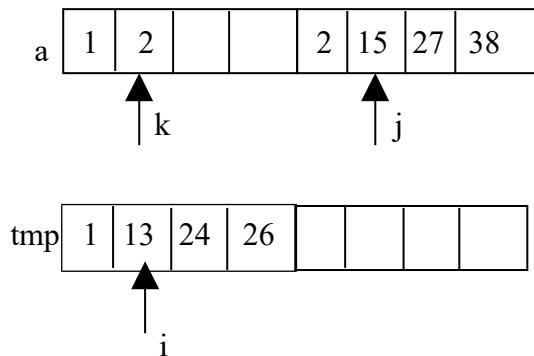
Stap 1.

Kopieer de linkerhelft van a in 'tmp'



Meng 'tmp' en rechterhelft van a





Er worden drie wijzers gebruikt: `i`, `j` en `k`

Als `tmp[i] <= a[j]`
 dan `a[k] = tmp[i]`, `k += 1`, `i += 1`
 anders `a[k] = a[j]`, `k += 1`, `j += 1`

```
def rmergeSort(a, low, high):
    if low < high:
        mid = (high+low)//2

        rmergeSort(a, low, mid)
        rmergeSort(a, mid+1, high)

        # kopieer linker helft in tmp
        tmpar = a[low:mid+1]

        # meng tmp en rechterhelft

        i = low
        j = mid+1
        k = low

        while i <= mid and j <= high:
            if tmpar[i-low] < a[j]:
                a[k] = tmpar[i-low]
                i += 1
            else:
                a[k] = a[j]
                j += 1
            k += 1
        while i <= mid:
            a[k] = tmpar[i-low]
            i += 1
            k += 1
```

Mergesort halveert bij elke recursieve aanroep de rij.

Het aantal halveringen bij n elementen is $2\log(n)$

Bij iedere halvering moeten n elementen vergeleken en zo nodig verplaatst worden.

Het algoritme is daarom $O(n \log(n))$

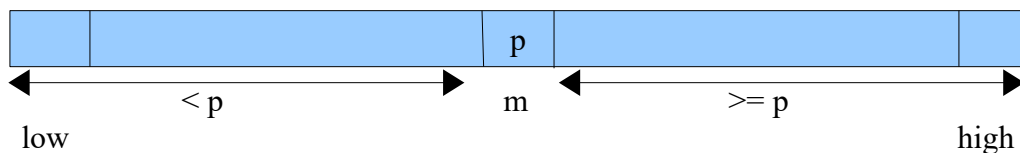
Een nadeel van Mergesort is het gebruik van een extra rij.

Het voordeel van Quicksort is, dat er geen extra rij nodig is.

Quicksort.

Bij quicksort werkt als volgt:

- kies een willekeurig pivot p
- plaats alle elementen, die kleiner dan p zijn, voor p
- plaats alle elementen, die groter dan p of gelijk aan p zijn, achter p
- pas quicksort toe op de deelrij links van p
- pas quicksort toe op de deelrij rechts van p



```
def swap(a,i,j):
    a[i],a[j] = a[j],a[i]

import random

def qsort(a,low=0,high=-1):
    if high == -1:
        high = len(a) -1
    if low < high:
        swap(a,low, random.randint(low,high))
        m = low
        for j in range(low+1,high+1):
            if a[j] < a[low]:
                m += 1
                swap(a,m,j)
            # low < i <= m : a[i] < a[low]
            # i > m       : a[i] >= a[low]
        swap(a,low,m)
        # low <= i < m : a[i] < a[m]
        # i > m       : a[i] >= a[m]
    if m > 0:
        qsort(a,low,m-1)
        qsort(a,m+1,high)
```

Quicksort is gemiddeld $O(n \log(n))$, maar is niet stabiel. Als de pivot steeds ongelukkig wordt gekozen is Quicksort $O(n^2)$. Dit komt echter zelden voor.

Backtracking.

Bij backtracking wordt systematisch een aantal mogelijkheden doorlopen. Als een gemaakte keuze niet blijkt te voldoen, wordt teruggekeerd naar de situatie voordat de keuze is gemaakt.

Een voorbeeld: het acht koninginnen probleem.

Zie http://nl.wikipedia.org/wiki/Acht_koninginnenprobleem .

Hoe worden 8 koninginnen op een schaakbord geplaatst, waarbij ze elkaar niet kunnen slaan?

Analyse van het probleem:

Ga uit van de volgende situatie: er zijn reeds vier koninginnen geplaatst.

	0	1	2	3	4	5	6	7
0				X				
1						X		
2								X
3			X					
4								
5								
6								
7								

De koninginnen staan op de posities (0,3), (1,5), (2,7) en (3,2)

De opstelling wordt vastgelegd door de lijst van kolomposities: [3,5,7,2]

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								

Als een koningin op positie (3,2) staat, zijn de rood gemarkeerde posities niet toegestaan.

Als op (x_0, y_0) en (x_1, y_1) een koningin staat dan moet het volgende gelden:

1. $x_0 \neq x_1$: niet op dezelfde rij
2. $y_0 \neq y_1$: niet op dezelfde kolom
3. $y_0 - x_0 \neq y_1 - x_1$: niet op dezelfde diagonaal (van linksboven naar rechtsonder)
4. $y_0 + x_0 \neq y_1 + x_1$: niet op dezelfde diagonaal (van rechtsonder naar linksboven)

Door een lijst van kolomposities bij te houden wordt aan voorwaarde 1 automatisch voldaan.
De voorwaarden 2 t/m 4 moeten worden gecheckt.

Het algoritme:

```
def check(a,i): # ga na of i aan a toegevoegd kan worden
    n = len(a)
    return not (i in a or
                # niet in dezelfde kolom
                i+n in [a[j]+j for j in range(n)] or
                # niet op dezelfde diagonaal
                i-n in [a[j]-j for j in range(n)])
                # niet op dezelfde diagonaal

def printQueens(a):
    n = len(a)
    for i in range(n):
        for j in range(n):
            if a[i] == j:
                print("X",end= " ")
            else:
                print("*",end= " ")
        print()
    print()

def rsearch(N):
    global a
    for i in range(N):
        if check(a,i):
            a.append(i)
            if len(a) == N:
                return True # geschikte a gevonden
            else:
                if rsearch(N):
                    return True
            del a[-1] # verwijder laatste element
    return False

a = [] # a geeft voor iedere rij de kolompositie aan
t = 0

rsearch(8)
print(a)
printQueens(a)
```

Uitvoer:

```
[0, 4, 7, 5, 2, 6, 1, 3]
X * * * * *
* * * * X * *
* * * * * * X
* * * * * X *
* * X * * * *
* * * * * X *
* X * * * * *
* * * X * * * *
```

Op de site http://rosettacode.org/wiki/N-queens_problem staat een korter algoritme:

```
from itertools import permutations

n = 8
cols = range(n)
for vec in permutations(cols):
    if n == len(set(vec[i]+i for i in cols)) \
        == len(set(vec[i]-i for i in cols)):
    print (vec )
```

Bij dit algoritme worden alle permutaties doorlopen en wordt per permutatie nagegaan of aan de voorwaarde is voldaan. Dit is een "brute force"-methode. Het algoritme is niet efficiënt. (Probeer n = 12)

Een overzicht:

n	aantal oplossingen
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2680
12	14200
13	73712

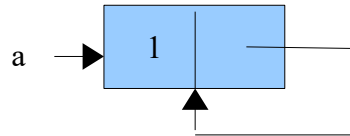
Kijk ook eens naar de site

http://www.animatedrecursion.com/advanced/the_eight_queens_problem.html

7. Recursieve datastructure.

Datastructuren kunnen naar zichzelf verwijzen.

Een voorbeeld:

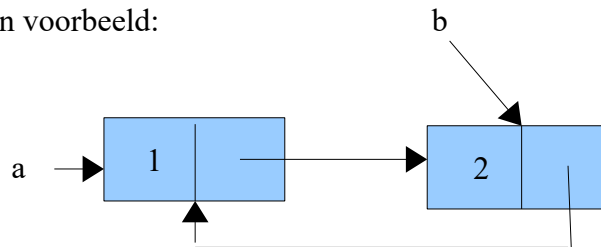


```
a = [1]
a.append(a)
print(a)
```

Uitvoer:

```
[1, [...]]
```

Nog een voorbeeld:



```
a = [1]
b = [2, a]

print(a)
print(b)

a.append(b)

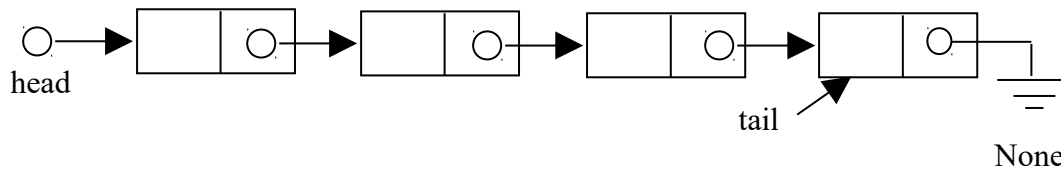
print(a)
print(b)
```

Uitvoer:

```
[1]
[2, [1]]
[1, [2, [...]]]
[2, [1, [...]]]
```

Lijst-structuren.

Bij Python speelt een lijst een belangrijke rol.
We laten nu zien hoe je zelf een lijst kunt maken.



Bij de implementatie worden twee klassen gebruikt:

een ListNode-klasse:

```
class ListNode:
    def __init__(self, data, next_node):
        self.data = data
        self.next = next_node

    def __repr__(self):
        return str(self.data)
```

en een List-klasse:

```
class MyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def __repr__(self):
        s = ''
        current = self.head
        if current != None:
            s = s + str(current)
            current = current.next
        while current != None:
            s = s + " -> " + str(current)
            current = current.next
        if not s: # s == '':
            s = 'empty List'
        return s

    def addLast(self, e):
        if not self.head: # self.head == None:
            self.head = ListNode(e, None)
            self.tail = self.head
        else:
            n = ListNode(e, None)
            self.tail.next = n
            self.tail = self.tail.next
```



```

def delete(self,e):
    if self.head: # self.head != None:
        if self.head.data == e:
            self.head = self.head.next
            if self.head == None:
                self.tail = None
        else:
            current = self.head
            while current.next != None and
                current.next.data != e:
                current = current.next
            if current.next != None:
                current.next = current.next.next
            if current.next == None:
                self.tail = current

```

Het programma

```

mylist = MyLinkedList()
print(mylist)
mylist.addLast(1)
mylist.addLast(2)
mylist.addLast(3)
print(mylist)
mylist.delete(2)
print(mylist)
mylist.delete(1)
print(mylist)
mylist.delete(3)
print(mylist)

```

heeft de volgende uitvoer:

```

empty list
1 -> 2 -> 3
1 -> 3
3
empty list

```

Voordelen van LinkedList:

- flexibel toevoegen van een nieuwe node
- flexibel verwijderen van een node

Voor het implementeren van een queue wordt de LinkedList-klasse gebruikt.

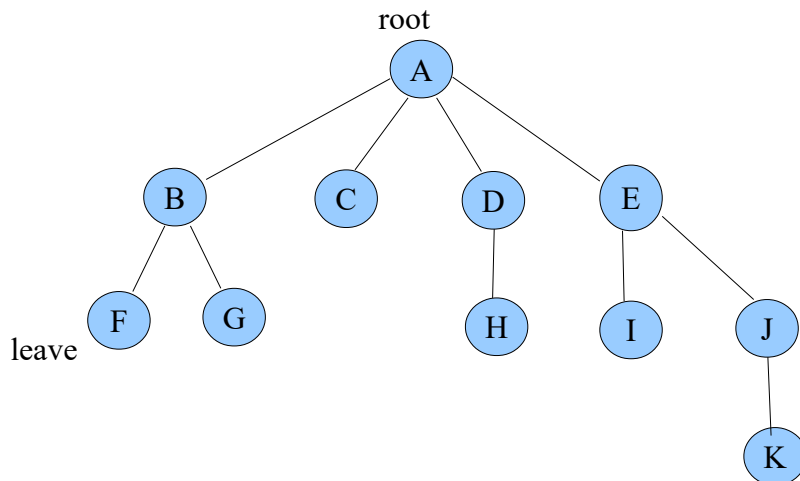
Nadelen van LinkedList:

- extra geheugenruimte nodig voor wijzers
- de nodes kunnen alleen bereikt worden door vanaf head de lijst te doorlopen. ($O(n)$)

8. Boom-structureën.

Een algemene boom heeft de volgende eigenschappen:

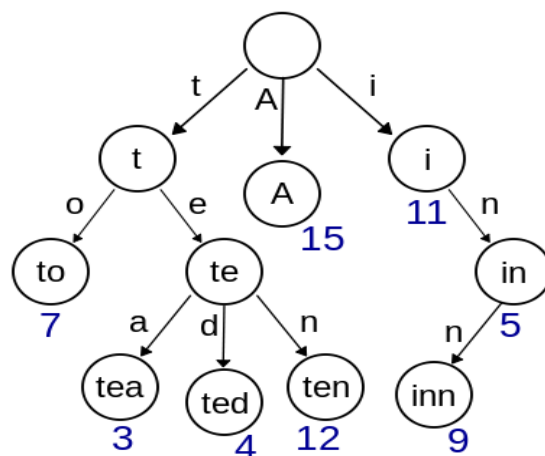
- één node is de root
- iedere node behalve de root heeft precies één parent
- iedere node heeft 0, 1 of meer kinderen
- een verbinding tussen twee nodes heet een tak (edge)
- het aantal takken van een pad heet de padlengte
- een eindnode heet een blad (leaf)



Lengte van het pad van A tot K : 3

Toepassingen:

- directory structuur in operating systems
- trie: opslag van woordenlijsten.
(zie <http://en.wikipedia.org/wiki/Trie>)



A trie for keys "A", "to", "tea", "ted", "ten", "i", "in", and "inn".

- XML-tree

Voorbeeld: toon de inhoud van een directory.

In Python:

```
import os

def treewalk(root, nspaces=0):
    # nspaces : aantal spaties dat ingesprongen wordt
    a = os.listdir(root)
    for f in a:
        print(' '*nspaces,end = '')
        if os.path.isdir(root + '/' + f):
            print('[ ' + f + ' ]')
            treewalk(root + '/' + f,nspaces+3) # recursieve aanroep
        else:
            print(f)

    root = os.getcwd()
    treewalk(root + '/../..')
```

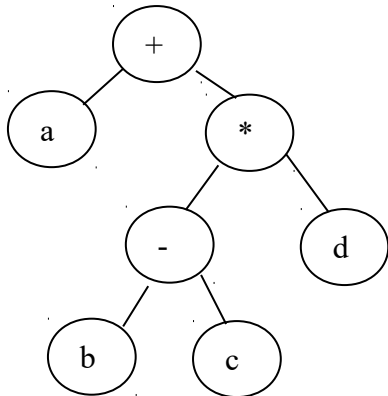
Toelichting:

<code>os.listdir(<dir>)</code>	: geef de listing van <dir>
<code>os.path.isdir(<name>)</code>	: ga na of <name> een directory is
<code>os.getcwd()</code>	: geef 'current work directory'

Binaire bomen.

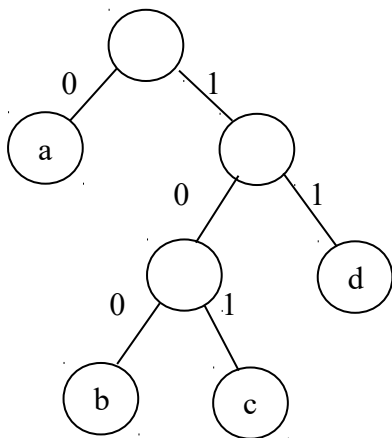
Voorbeelden van binaire bomen zijn:

Een expression tree: $a + ((b - c) * d)$



Een expression tree wordt door de compiler gebruikt voor het berekenen van de expressie. De boom legt de volgorde van de berekening vast.

Een 'huffman coding tree'.



Het bitpatroon 10101000 correspondeert met de reeks caba
c ab a

Op deze manier kunnen 4 karakters door 1 byte worden vastgeelgd.

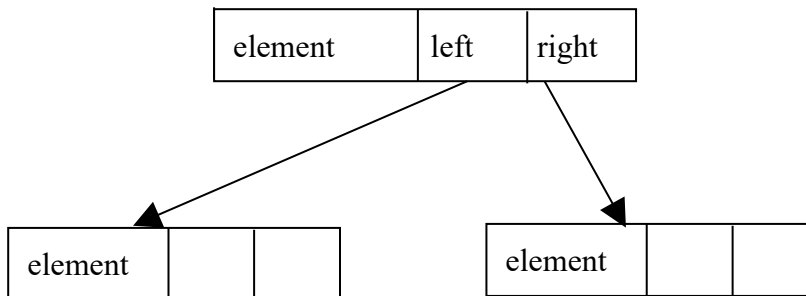
Huffman coding trees worden gebruikt voor het comprimeren van data.

Binaire bomen zijn opgebouwd uit nodes.

Een node bevat de volgende attributen:

- element
- left
- right

Een node is als volgt weer te geven:



Binaire bomen worden vaak geïmplementeerd met twee klassen:

- een recursieve BinaryNode-klasse
- een BinaryTree-klasse. Deze klasse bevat één attribuut: root.

In Python:

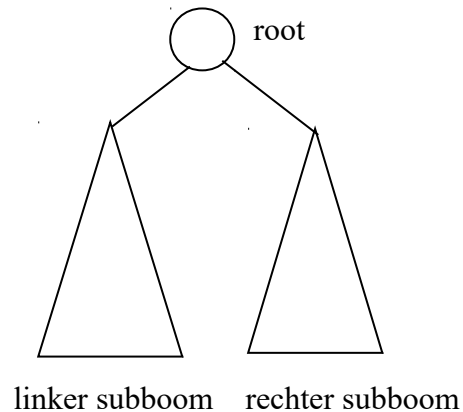
```
class BinaryNode:
    def __init__(self, element=None, left=None, right=None):
        self.element = element
        self.left = left
        self.right = right

class BinaryTree:
    def __init__(self, root=None):
        self.root = root
```

Van een boom kunnen we o.a. de volgende gegevens opvragen:

- size: het aantal nodes
- height: de padlengte van het pad tussen de root en de diepste leaf

Deze gegevens kunnen berekend worden m.b.v. een recursieve algoritme:



Berekening size:

- $\text{size}(\text{None}) = 0$
- $\text{size}(\text{node}) = 1 + \text{size}(\text{linker subboom}) + \text{size}(\text{rechter subboom})$

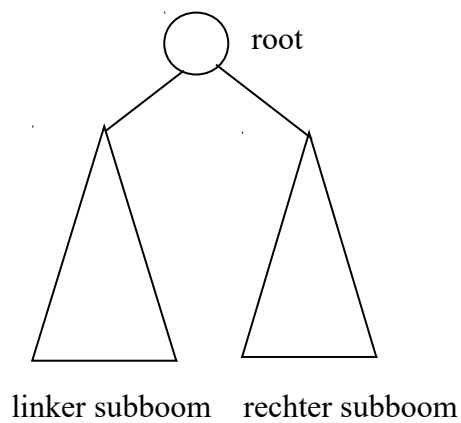
In Python:

class BinaryNode:

```
def size(self):
    s = 1
    if self.left:
        s += self.left.size()
    if self.right:
        s += self.right.size()
    return s
```

class BinaryTree:

```
def size(self):
    if self.root:
        return self.root.size()
    else:
        return 0
```



Berekening height:

- $\text{height}(\text{None}) = -1$;
- $\text{height}(\text{node}) = 1 + \max(\text{height}(\text{linker subboom}), \text{height}(\text{rechter subboom}))$

In Python:

class BinaryNode:

```

def height(self):
    if self.left:
        hleft = self.left.height()
    else:
        hleft = -1
    if self.right:
        hright = self.right.height()
    else:
        hright = -1
    return 1 + max(hleft, hright)

```

class BinaryTree:

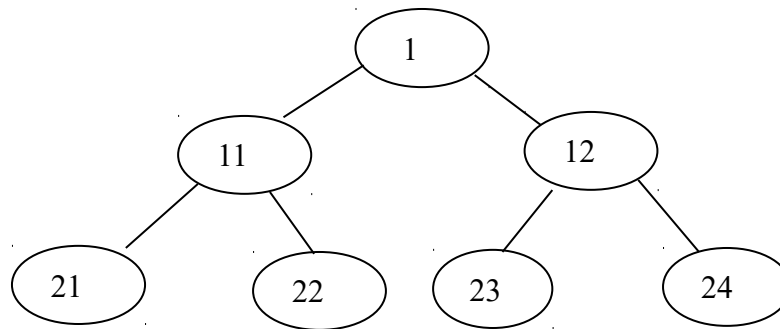
```

def height(self):
    if self.root:
        return self.root.height()
    else:
        return -1

```

Een boom opbouwen.

We zullen de volgende boom opbouwen:



Dit gaat m.b.v. de volgende twee functies:

```
def make_node(a):  
    if not a:  
        return None  
    if len(a) == 1:  
        return BinaryNode(a[0],None,None)  
    mid = len(a)//2;  
    left_node = make_node(a[:mid])  
    right_node = make_node(a[mid+1:])  
    return BinaryNode(a[mid],left_node,right_node)  
  
def make_tree(a):  
    root = make_node(a)  
    return BinaryTree(root)
```


Een boom afdrukken.

Het is eenvoudiger een boom gekanteld af te drukken:

In Python:

class BinaryNode:

```
def __repr__(self, nspaces=0):
    s1 = ''
    if self.right:
        s1 = self.right.__repr__(nspaces + 3)
    s2 = ' '*nspaces + str(self.element) + '\n'
    s3 = ''
    if self.left:
        s3 = self.left.__repr__(nspaces + 3)
    return s1 + s2 + s3
```

class BinaryTree:

```
def __repr__(self):
    if self.root:
        return str(self.root)
    else:
        return 'null-tree'
```

De boom kan er als volgt uit zien:

```

      15
     14
    13
   12
  11
 10
 9
8
 7
 6
 5
4
 3
 2
 1
```

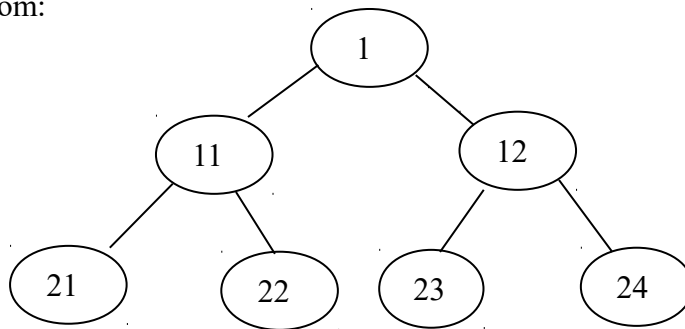
Een boom doorlopen.

Een boom kan op de verschillende manieren doorlopen worden:

1. preorder: eerst root, dan linker subboom, dan rechter subboom
2. postorder: eerst linker subboom, dan rechter subboom, dan root
3. inorder: eerst linker subboom, dan root, dan rechter subboom
4. level order, eerst de root, dan de kinderen van de root, etc

Vb.

Bij de boom:



is de volgorde waarin de nodes doorlopen worden als volgt:

preorder	: 1 11 21 22 12 23 24
inorder	: 21 11 22 1 23 12 24
postorder	: 21 22 11 23 24 12 1
level order	: 1 11 12 21 22 23 24

9. Binary Search Trees.

Lijsten : $O(n)$

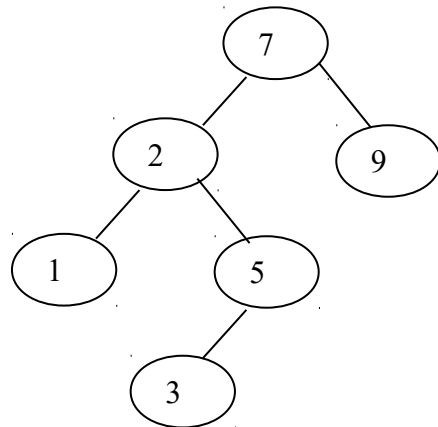
Binaire zoekbomen: $O(\log(n))$

Voor iedere node x geldt:

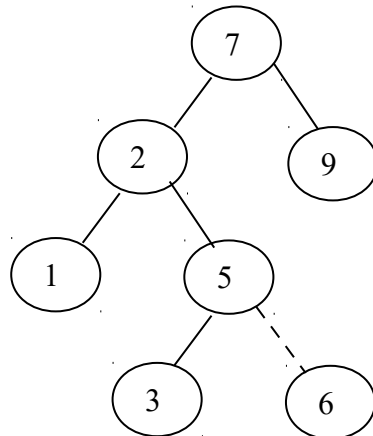
de key van een node in de linker subtree van x is kleiner dan de key van x

de key van een node in de rechter subtree van x is groter dan de key van x

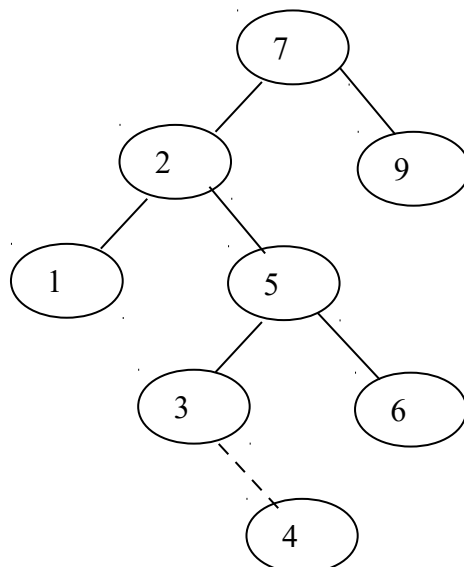
Vb.



Toevoegen van 6:



Toevoegen van 4:



Een element mag maar één keer toegevoegd worden.

Zoeken in een binaire boom.

class BSTNode:

(niet recursief)

```
def search(self,e):
    current = self
    found = False
    while not found and current:
        if current.element < e:
            current = current.right
        elif current.element > e:
            current = current.left
        else:
            found = True
    if found:
        return current
    else:
        return None
```

class BST:

```
def search(self,e):
    if self.root and e:
        return self.root.search(e)
    else:
        return None
```

Een element toevoegen.

class BSTNode:

(niet recursief)

```
def insert(self,e):
    parent = self
    current = None
    found = False

    if parent.element < e:
        current = parent.right
    elif parent.element > e:
        current = parent.left
    else:
        found = True;

    while not found and current:
        parent = current
        if parent.element < e:
            current = parent.right
        elif parent.element > e:
            current = parent.left
        else:
            found = True

    if not found:
        if parent.element < e:
            parent.right = BSTNode(e,None,None)
        else:
            parent.left = BSTNode(e,None,None)
    return not found
```

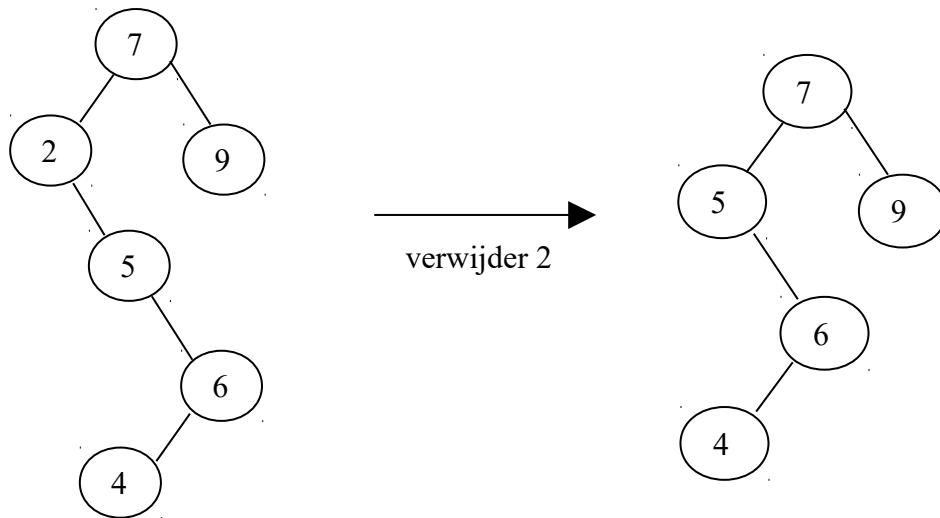
class BST:

```
def insert(self,e):
    if e:
        if self.root:
            return self.root.insert(e)
        else:
            self.root = BSTNode(e,None,None)
            return True
    return False
```

Een element verwijderen.

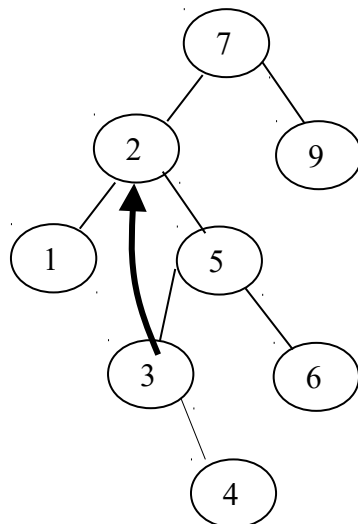
Het verwijderen van een leaf is geen probleem.

Het verwijderen van een interne node met 1 kind gaat ook eenvoudig.



Het verwijderen van een interne node met 2 kinderen gaat als volgt:

Verwijder 2

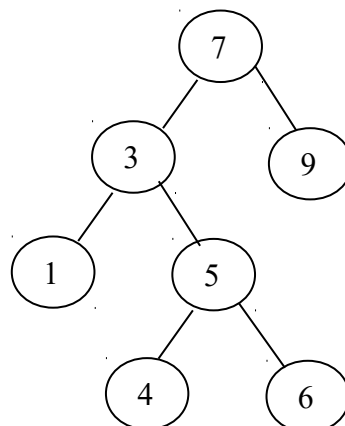


Bepaal de kleinste waarde van de rechtersubboom van node 2

Vervang node 2 door node 3

De rechtersubboom van oorspronkelijke node 3 wordt nu de linkersubboom van de ouder van de oorspronkelijke node 3.

Dit levert het volgende op:



10. Gebalanceerde bomen.

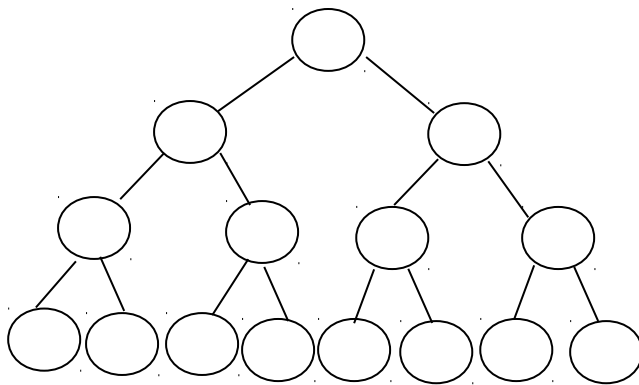
Tussen het aantal nodes van een boom en de hoogte van de boom bestaat de volgende relatie:

Als de hoogte n is dan geldt: $n+1 \leq \text{aantal nodes} \leq 2^{n+1}-1$

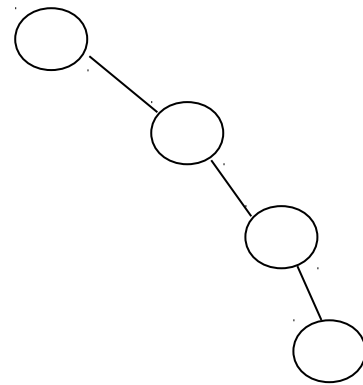
Als het aantal nodes k is, dan geldt $\lceil \log k \rceil \leq \text{hoogte} \leq k-1$

Als we willen dat toevoegen, zoeken en verwijderen van een node in een binaire zoekboom $O(\log(n))$ is dan moet de hoogte van de boom $O(\log(n))$ zijn

Een boom, die hieraan voldoet, heet een gebalanceerde boom.

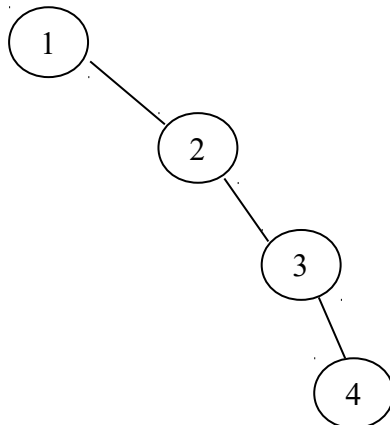


Gebalanceerde boom



Niet gebalanceerde boom

Voegen we achtereenvolgens aan een lege binaire zoekboom de volgende nodes toe 1,2,3,4 dan krijgen we de volgende boom:



De boom is dus niet gebalanceerd.

Dit kan opgelost worden door de volgende technieken:

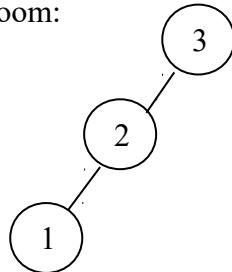
- AVL-trees (zie http://en.wikipedia.org/wiki/AVL_tree)
- red-black tree (zie http://en.wikipedia.org/wiki/Red%E2%80%93black_tree)

De AVL-tree.

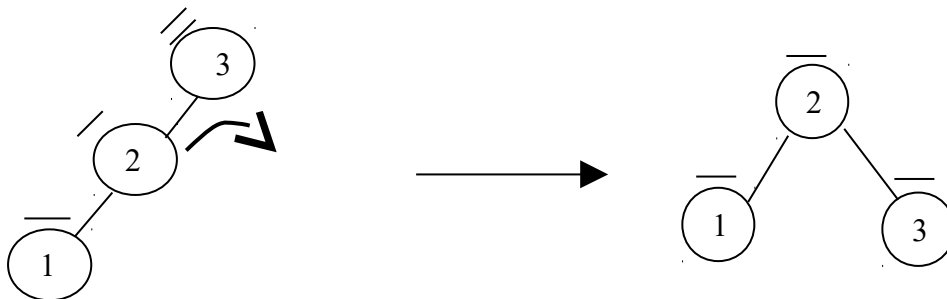
Eis: bij iedere node mag het hoogteverschil tussen de linker-subboom en de rechter-subboom ten hoogste 1 zijn.

Voorbeeld: ga uit van een lege boom en voeg achtereenvolgens toe: 3 2 1

We krijgen dan de volgende boom:



Deze boom voldoet niet aan de eis. Dit wordt opgelost door een 'single rotation'

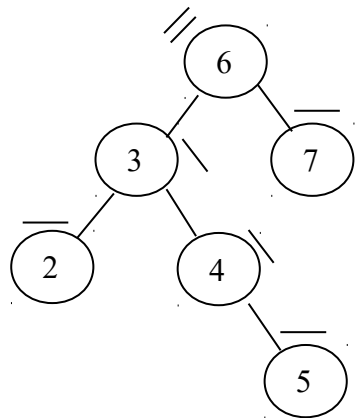


// : linker subboom is meer dan 1 hoger dan rechter subboom
het verschil in hoogte is te groot

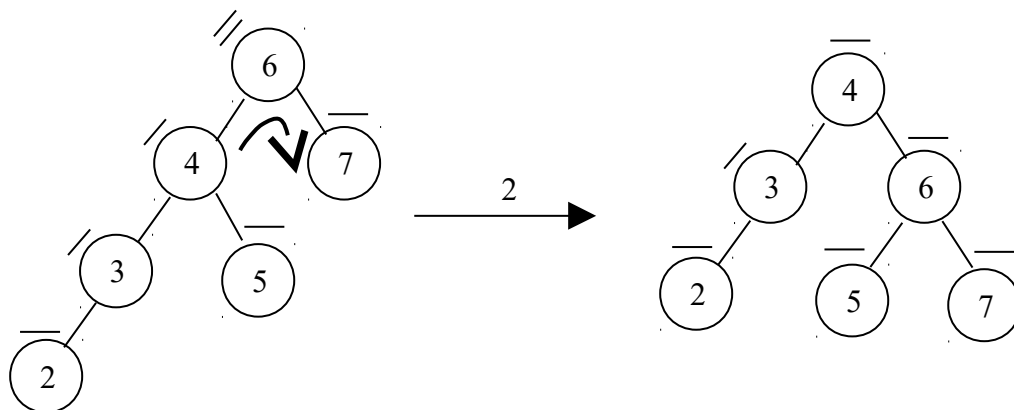
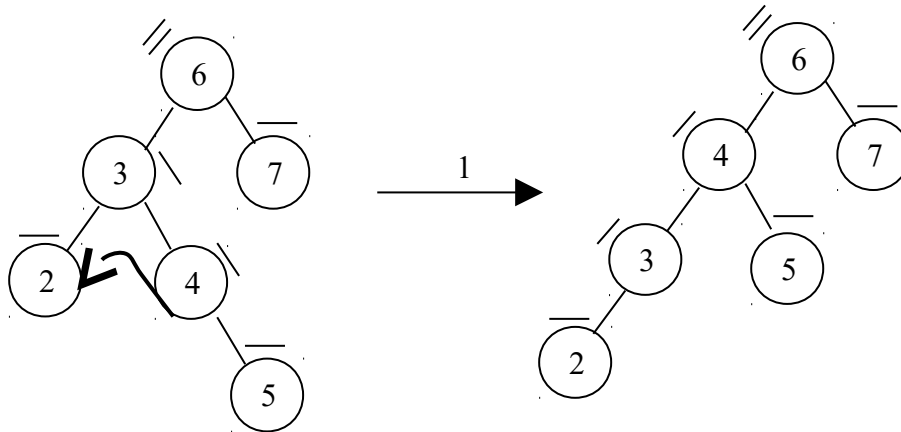
/ : linker subboom is 1 hoger dan rechter subboom

— : linker subboom en rechter subboom zijn even hoog

Ga uit van een lege boom en voeg de volgende nodes toe: 6 3 7 2 4 5



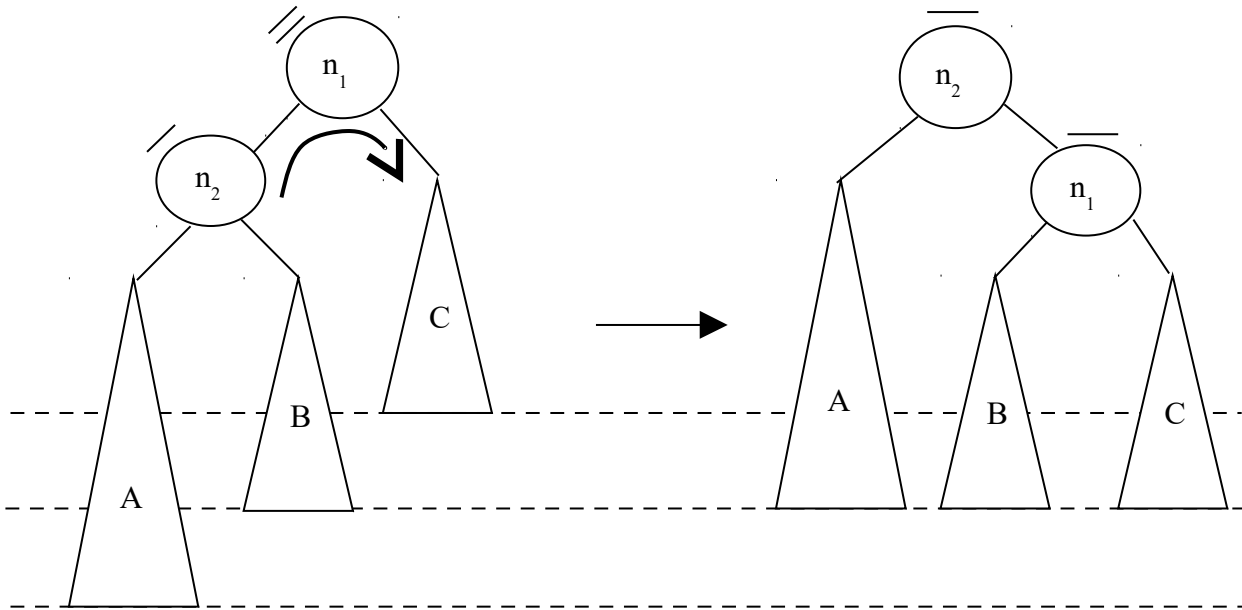
Deze boom voldoet niet aan de eis. Dit wordt opgelost door een 'double rotation'



Single rotation: dubbele streep en streep van node staan in elkaars verlengde

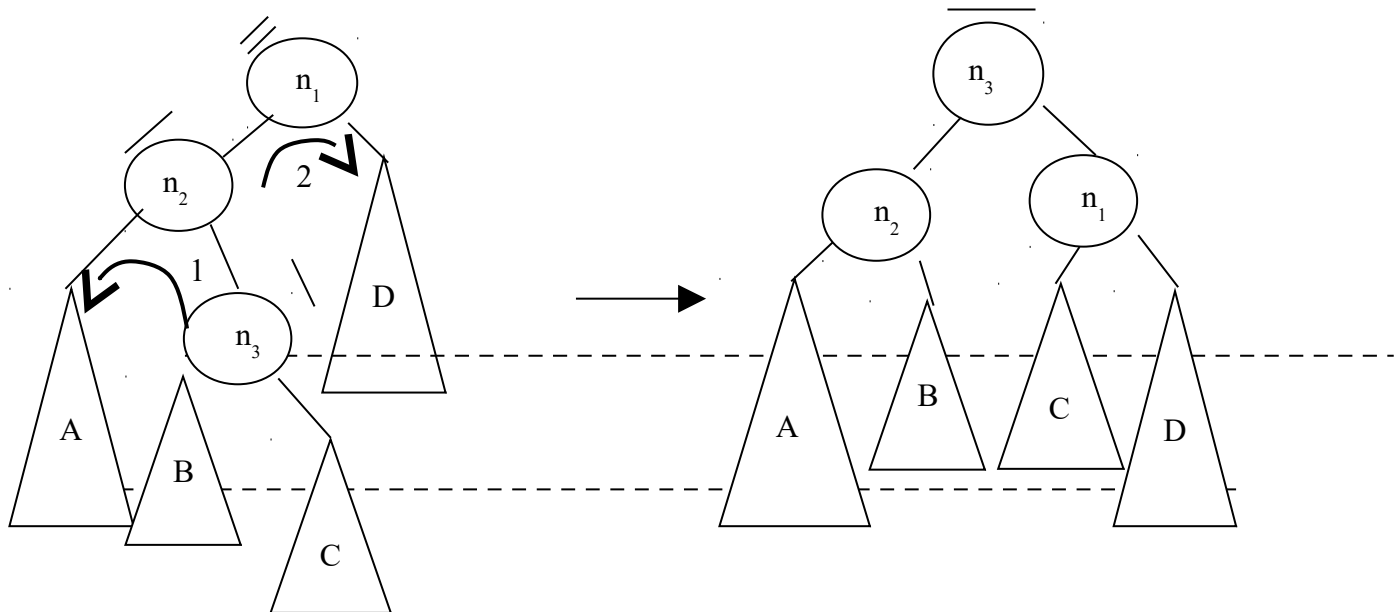
Double rotation: dubbele streep en streep van node staan niet in elkaars verlengde

Algemene structuur bij 'single rotation'



Door een toevoeging van een node in subtree A is de balans verstoord.. Na de rotatie zijn de balans en de hoogte hersteld.

Algemene structuur bij 'double rotation'



Door een toevoeging van een node in subtree B of C is de balans verstoord.. Na de rotatie zijn de balans en de hoogte hersteld.

Door toevoeging van een node kunnen er op het pad tussen de root en de toegevoegde node meerdere balansverstoringen optreden. Door de balans zo dicht mogelijk bij de toegevoegde node te herstellen zijn automatisch de overige balansverstoringen hersteld. Dit komt omdat de hoogte van de subtree waar de node aan toegevoegd is de oorspronkelijke waarde heeft teruggekregen.

Merk op dat de subtrees B en C na de rotatie aan andere nodes zijn gekoppeld.

Red black tree.

Een red black tree is ook een gebalanceerde boom.

Eigenschappen van een red-black tree:

- iedere node is rood of zwart
- de root-node is zwart
- een kind van een rode node is zwart
- het aantal zwarte nodes van ieder pad van root tot None-leaf is gelijk

Demo:

<http://www.cs.uc.edu/~franco/C321/html/RedBlack/redblack.html>

Toevoegen:

De toegevoegde node heeft de kleur rood.

Als de vader-node ook rood is, dan moet de boom aangepast worden.

Aanpassingen zijn realiseren door kleuren te wijzigen en door rotaties.

Verwijderen:

Als de verwijderde node zwart was, dan moet de boom aangepast worden.

11. Hashing.

Binaire zoekbomen: $O(\log(n))$

Hashing : $O(1)$, m.a.w. gemiddeld 2 stappen

Bij Python is een dictionary gebaseerd op hashing.

Hashing kent de volgende onderdelen:

- een hash-table
- een hash-functie
- een collision strategie (collision = botsing)

Elementen worden opgeslagen in een hash-table:

```
len = ...  
table = [None]*len
```

Een hashentry bevat het element en extra informatie.

De positie van een item wordt bepaald d.m.v. een hashfunctie.

Voorbeeld:

Ga uit van items, met keys 14, 78, 34, 79, 43

Kies hash-table: $[None]*100$
 hashfunctie: $\text{hash}(\text{key}) = \text{key}$

Het nadeel van deze methode is dat de tabel voor een klein gedeelte gebruikt wordt.

We kunnen een kleiner tabel nemen:

Kies hash-table: [None]*7
 hashfunctie: $\text{hash}(\text{key}) = \text{key} \% 7$;

Dit levert de volgende tabel op

key	hash(key)
14	0
78	1
34	6
79	2
43	1

Hashtable

0	14	botsing (collision)
1	78 43	
2	79	
3		
4		
5		
6	34	

De hashtable is nu beter gevuld.

Er is echter sprake van een botsing.

Een goede hash-functie zorgt voor zo weinig mogelijk botsingen.
De hash-waarden moeten zoveel mogelijk over de hashtable verspreid worden.

De kans op een botsing blijft.

We behandelen de volgende strategieën voor het oplossen van botsingen:

- linear hashing
- quadratic hashing
- separate chaining hashing

Linear hashing:

Methode: Bepaal positie $p = \text{hash}(\text{key})$.
 Onderzoek achtereenvolgens de posities

p
 $(p+1) \% \text{hashSize}$ (de buren)
 $(p+2) \% \text{hashSize}$ (de buren van de buren)
enz.

totdat een vrije positie gevonden is.

Vb: $\text{tableSize} = 10$
 $\text{hash}(\text{key}) = \text{key} \% 10$;

Voeg de volgende items toe: 89, 18, 49, 58, 9

0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Nadeel bij linear hashing: grote kans op clustering van keys.
Hierdoor neemt de performance af.

Het aantal posities dat onderzocht moet worden hangt af van de vullingsgraad van de hashtabel.

$$\text{Vullingsgraad (load factor) } \lambda = \frac{\text{aantal items}}{\text{hashSize}}$$

Er geldt: $0 \leq \lambda \leq 1$

Bij linear hashing geldt het volgende (bij benadering):

	aantal onderzochte posities	$\lambda = 0$	$\lambda = 0.5$	$\lambda = 0.9$	$\lambda = 1$
Toevoegen	$\frac{1}{2} \left(1 + \frac{1}{(1-\lambda)^2}\right)$	1	2.5	50.5	∞
Zoeken zonder succes	$\frac{1}{2} \left(1 + \frac{1}{(1-\lambda)^2}\right)$	1	2.5	50.5	∞
Zoeken met succes	$\frac{1}{2} \left(1 + \frac{1}{1-\lambda}\right)$	1	1.5	5.5	∞

Conclusie: bij hoge vullingsgraad neemt door clustering de performance aanzienlijk af.

Het verwijderen is een apart probleem. Wat gebeurt er als in bovenstaand voorbeeld 49 wordt verwijderd?

Mogelijke oplossingen:

- op de positie een speciale item plaatsen, bijvoorbeeld -1
- per positie een boolean bijhouden, waarbij wordt aangegeven dat de positie nu vrij is, maar in een eerder stadium is gebruikt.

Implementatie-details

HashEntry-object:

```
class HashEntry:
    def __init__(self, element, isActive):
        self.element = element
        self.isActive = isActive
```

Als het element wordt verwijderd dan krijgt isActive de waarde False.
Bij het zoeken van een element wordt hier gebruik van gemaakt.

Search:

```
def getPos(self, e):
    current = hash(e) % self.len
    while self.table[current] != None and \
        self.table[current].element != e:
        current = (current + 1) % self.len
    return current

def search(self, e):
    current = self.getPos(e)
    if self.table[current] != None and \
        self.table[current].isActive:
        return current
    else:
        return -1
```


Insert

```
def insert(self,e):
    current = self.getPos(e)
    if self.table[current] == None:
        self.table[current] = HashEntry(e,True)
        self.size += 1
        self.used += 1
    elif not self.table[current].isActive:
        self.table[current].isActive = True
        self.size += 1
    else:
        current = -1
    if self.used > 0.8*self.len:
        self.rehash(4*self.size)
    return current
```

size: aantal actieve elementen

used: aantal actieve en niet-actieve elementen

Als de vullingsgraad meer dan 80% is dan vindt rehashing plaats.

Delete

```
def delete(self,e):
    current = self.search(e)
    if current != -1:
        self.table[current].isActive = False
    self.size -= 1
    return current
```

Rehash

```
def rehash(self,newLen):
    if newLen >= self.size:
        oldLen = self.len
        oldTable = self.table

        self.size = 0
        self.used = 0
        self.len = newLen
        self.table = [None]*self.len

    for i in range(oldLen):
        if oldTable[i] != None and oldTable[i].isActive:
            self.insert(oldTable[i].element)
```

Quadratic hashing

Quadratic hashing lost het clustering-probleem als volgt op:

Methode: Bepaal positie $p = \text{hash}(\text{key})$.
 Onderzoek achtereenvolgens de posities

p
 $(p+1) \% \text{hashSize}$ (de burens)
 $(p+4) \% \text{hashSize}$ (verder dan de burens van de burens)
 $(p+9) \% \text{hashSize}$
enz.

totdat een vrije positie gevonden is.

Door de afstand tussen twee opeenvolgende posities te vergroten, wordt clustering vermeden.

Vb: $\text{tableSize} = 10$
 $\text{hash}(\text{key}) = \text{key} \% 10$;

Voeg de volgende items toe: 89, 18, 49, 58, 9

0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Nu bestaat het risico, dat in een cyclus steeds dezelfde posities worden onderzocht, terwijl er nog genoeg lege posities zijn.

Om dit te voorkomen moet aan de volgende voorwaarden worden voldaan:

- de hashSize moet een priemgetal zijn
- de load factor moet dus $\leq \frac{1}{2}$ zijn. (Dit wordt gerealiseerd door de grootte van de hashtable te vergroten zodra de load factor te groot dreigt te worden.)

Implementatie-details.

Bij linear hashing:

```
def getPos(self,e):
    current = hash(e)%self.len
    while self.table[current] != None and \
           self.table[current].element != e:
        current = (current+1) % self.len
    return current
```

Bij quadratic hashing:

```
def getPos(self,e):
    i = 0
    current = hash(e)%self.len
    while self.table[current] != None and \
           self.table[current].element != e:
        current = (current+2*i+1) % self.len
        i += 1
    return current
```

Bij insert:

```
if 2*self.used > self.len:
    self.rehash(nextPrime(4*self.size))
```

Het bepalen van een priemgetal gaat als volgt: (is 1 een priemgetal?)

```
def isPrime(n):
    if n<=0:
        return False
    elif n<=2:
        return True
    elif n>2 and n%2==0:
        return False
    else:
        i = 3;
        while i*i <= n and n%i != 0:
            i +=2
        return i*i > n

def nextPrime(n):
    if n <= 0:
        return 1
    i = n
    while not isPrime(i):
        i += 1
    return i;
```

Separate chaining hashing.

Bij deze methode worden items, die bij een positie behoren, in een lijst te plaatsen.

Vb: tableSize = 10
 hash(key) = key%10;

Voeg de volgende items toe: 89, 18, 49, 58, 9

0	
1	
2	
3	
4	
5	
6	
7	
8	18 58
9	89 49 9

HashSet en HashMap van Java zijn hierop gebaseerd.

Het voordeel hiervan is dat geen overflow kan plaatsvinden. De load factor moet maximaal 0.75 zijn.

Hashfuncties.

Python bevat de functie hash.

De volgende regels gelden:

- de hash-waarde is een geheel getal i met $-2^{31} \leq i \leq 2^{31}-1$ (bij een 32-bits systeem)
- als $x1 == x2$ dan is $\text{hash}(x1) == \text{hash}(x2)$

Als $\text{hash}(x1) == \text{hash}(x2)$ is dan ook $x1 == x2$?

Meestal wel, maar niet altijd!

Voorbeeld:

```
>>> hash(-1)
-2
>>> hash(-2)
-2
>>>
```

Toelichting: als $-2^{31} \leq i \leq 2^{31}-1$ dan is $\text{hash}(i) == i$

Er één uitzondering: $\text{hash}(-1) == -2$

Dit is gedaan, omdat -1 vaak als foutcode wordt gebruikt.

Bij gebroken getallen wordt de hash-waarde berekend door op bit-niveau shift- en xor-operaties uit te voeren.

Hierdoor krijg je bij een kleine afwijking een totaal andere hash-waarde

Voorbeeld:

gebroken getallen:

```
>>> hash(1.1)
2040142438
>>> hash(1.1001)
-1863016492
>>>
```

strings:

```
>>> hash('stringhash')
1779862525
>>> hash('stringhasi')
1779862524
>>> hash('stringhasj')
1779862527
>>> hash('ttringhash')
-348353928
>>> hash('stringhas')
-896990852
>>> hash('stringhasj')
1779862527
```

Bij strings geldt het volgende:

- als het laatste karakter wordt gewijzigd dan verandert de key-waarde niet veel
- als het laatste karakter wordt verwijderd, dan is de key-waarde volledig anders
- als het eerste karakter wordt gewijzigd, dan is de key-waarde volledig anders

Bij strings geldt ook de volgende eigenschap: de hashwaarde van een string is bij iedere Python-omgeving verschillend. Dit is gedaan ter bescherming tegen 'denial of service'-attacks. (zie https://docs.python.org/3.6/reference/datamodel.html#object.__hash__ en <http://www.ocert.org/advisories/ocert-2011-003.html>) Dit geldt ook voor bytes en datetime-objects.

Vergelijking tussen Hashtables en Binary Trees.

HashTables:

Voordeel: snelle toegang tot de elementen: $O(1)$

Nadeel: de elementen zijn niet gesorteerd

Binary Tree:

Voordeel: de elementen zijn gesorteerd

Nadeel: minder snelle toegang tot de elementen: $O(\log(n))$

12. Dynamisch programmeren.

Bij dynamisch programmeren worden tussen-resultaten bewaard.

Voorbeeld 1: De getallen van Fibonacci.

Deze zijn als volgt gedefinieerd:

$$\begin{aligned}F_0 &= 0 \\F_1 &= 1 \\F_n &= F_{n-1} + F_{n-2} \text{ als } n > 1\end{aligned}$$

Daarmee worden de volgende getallen verkregen: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Oplossing zonder recursie:

```
def fib(n):
    if n <= 1:
        return n
    else:
        a = [0]*(n+1)
        a[0],a[1] = 0,1
        print(a)
        for i in range(2,n+1):
            a[i] = a[i-1] + a[i-2]
            print(a)
        return a[n]
print(fib(10))
```

Uitvoer:

```
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 1, 2, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 1, 2, 3, 0, 0, 0, 0, 0, 0]
[0, 1, 1, 2, 3, 5, 0, 0, 0, 0, 0]
[0, 1, 1, 2, 3, 5, 8, 0, 0, 0, 0]
[0, 1, 1, 2, 3, 5, 8, 13, 0, 0, 0]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 0, 0]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 0]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
55
```


De driehoek van Pascal kun je ook met de "n boven k"- notatie weergeven:

$$\begin{array}{ccccccc}
 & & \binom{0}{0} & 1 & & & \\
 & & & & & & \\
 & \binom{1}{0} & 1 & & \binom{1}{1} & 1 & \\
 & & & & & & \\
 & \binom{2}{0} & 1 & & \binom{2}{1} & 2 & & \binom{2}{2} & 1 \\
 & & & & & & & & \\
 \binom{3}{0} & 1 & & \binom{3}{1} & 3 & & \binom{3}{2} & 3 & & \binom{3}{3} & 1 \\
 & & & & & & & & & & \\
 \binom{4}{0} & 1 & & \binom{4}{1} & 4 & & \binom{4}{2} & 6 & & \binom{4}{3} & 4 & & \binom{4}{4} & 1
 \end{array}$$

Op basis hiervan kun je $\binom{6}{3}$ als volgt berekenen:

regels : rij[0] = 1 voor iedere rij
 volgende rij [i] = rij[i] + rij[i-1]

i :	0	1	2	3
rij 0 :	1			
rij 1 :	1	1		
rij 2 :	1	2	1	
rij 3 :	1	3	3	1
rij 4 :	1	4	6	4
rij 5 :	1	5	10	10
rij 6 :	1	6	15	20

dus $\binom{6}{3} = 20$

Voorbeeld 3: hoe krijg je het bedrag € 87,73 betaald met zo weinig mogelijk munten en papiergeld?

Overzicht:

Euromunten: 1, 2, 5, 10, 20, 50, 100, 200 eurocent

Eurobiljetten: 5, 10, 20, 50, 100, 200, 500 euro

Strategie:

- Neem de munt of het biljet met de grootste waarde kleiner dan het bedrag.
- Trek de waarde van de munt of het biljet van het bedrag af
- Herhaal dit proces totdat 0 euro overblijft.

bedrag	munt
87,73	50 euro
$87,73 - 50 = 37,73$	20 euro
$37,73 - 20 = 17,73$	10 euro
$17,73 - 10 = 7,73$	5 euro
$7,73 - 5 = 2,73$	2 euro
$2,73 - 2 = 0,73$	50 eurocent
$0,73 - 0,50 = 0,23$	20 eurocent
$0,23 - 0,20 = 0,03$	2 eurocent
$0,03 - 0,02 = 0,01$	1 eurocent
$0,01 - 0,01 = 0$	

Een dergelijke aanpak wordt de Greedy algoritme methode genoemd.

Het werkt vaak, maar niet altijd.

Voorbeeld: een geldsysteem met de munten 1,5,10, 20 en 25.

Greedy-methode bij bedrag van 40:

Bedrag	munt
40	25
$40 - 25 = 15$	10
$25 - 10 = 5$	5
$5 - 5 = 0$	

Bij 40 komen we uit op drie munten 25,10 en 5

Met twee munten van 20 kunnen we echter ook volstaan: $40 = 20+20$

Het algoritme om dit te achterhalen gaat achtereenvolgens bij de bedragen 1,2 3, etc na hoeveel munten nodig zijn:

bedrag	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
aantal	1	2	3	4	1	2	3	4	5	1	2	3	4	5	2	3	4	5	6	1	2	3	4	5	1	2	3	4	5	2

Het aantal munten bij 29 wordt als volgt berekend:

Bepaal voor elke munt het minimum aantal munten dat nodig is voor $29 - \text{munt-waarde}$.

munt	$29 - \text{munt-waarde}$	aantal munten
1	28	$4+1$
5	24	$5+1$
10	19	$6+1$
20	9	$5+1$
25	4	$4+1$

Voor 29 zijn dus minimaal $4+1$ munten nodig: $1 + 1+ 1+1 + 25$

Voorbeeld 4: Hoe wordt een plank met lengte n zo rendabel mogelijk verzaagd?

lengte	1	2	3	4	5	6	7	8
prijs	1	5	8	9	10	17	17	20

Bij lengte n zijn er 2^{n-1} mogelijkheden.

Strategie:

$p[i]$ = "prijs lengte i "

$m[i]$ = "de maximale opbrengst bij lengte i "

Er geldt: $m[i] = \max(p[i], p[1] + m[i-1], p[2] + m[i-2], \dots, p[i-1] + m[1])$

Als we $m[4]$ willen berekenen, bepalen we achtereenvolgens $m[1]$, $m[2]$, $m[3]$ en $m[4]$

$m[1] = \max(p[1]) = \max(1) = 1$

$m[2] = \max(p[2], p[1] + m[1]) = \max(5, 1+1) = 5$

$m[3] = \max(p[3], p[1] + m[2], p[2] + m[1]) = \max(8, 1+5, 5+1) = 8$

$m[4] = \max(p[4], p[1] + m[3], p[2] + m[2], p[3] + m[1]) = \max(9, 1+8, 5+5, 8+1) = 10$

Voorbeeld 5: Het knapsack-probleem

Een dief breekt in en heeft een tas, waarin maximaal 13 kilo kan.
Hij vindt vier artikelen.

De volgende tabel geeft en overzicht van gewicht en prijs.

artikel	0	1	2	3
gewicht	3	4	6	8
prijs	4	5	8	11

Hoe kan hij de zak optimaal vullen?

Strategie:

$w[i]$ = gewicht artikel i

$b[i]$ = bedrag artikel i

In een matrix M wordt het volgende vastgelegd:

$M[i,j]$ = totaalbedrag beste selectie uit artikel 0 t/m artikel i met totaalgewicht $\leq j$

Berekening $M[i,j]$:

- $M[i,0] = 0$

$i = 0$:

- als $w[0] > j$ dan is $M[0,j] = 0$
- als $w[0] \leq j$ dan is $M[0,j] = w[0]$

$i > 0$:

- als $w[i] > j$ dan is $M[i,j] = M[i-1,j]$
- als $w[i] \leq j$ dan is $M[i,j] = \max(M[i-1,j], M[i-1,j-w[i]] + b[i])$

De matrix M kan rij na rij berekend worden.

artikel	w	b	j:	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	3	4		0	0	0	4	4	4	4	4	4	4	4	4	4	4
0,1	4	5		0	0	0	4	5	5	5	9	9	9	9	9	9	9
0,1,2	6	8		0	0	0	4	5	5	8	9	9	12	13	13	13	17
0,1,2,3	8	11		0	0	0	4	5	5	8	9	11	12	13	15	16	17

13. Grafen.

Inleiding.

Op allerlei gebieden spelen grafen een belangrijke rol. Denk hierbij aan netwerkinfrastructuren, routes binnen een netwerk en het regelen van netwerkverkeer.

In dit hoofdstuk zullen een aantal begrippen en algoritmen behandeld worden.

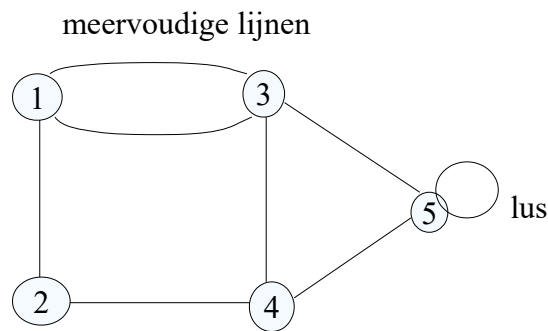
De behandelde algoritmen zijn:

- breadth first search
- depth first search
- het algoritme van Kruskal
- het algoritme van Prim
- het 'kortste pad'-algoritme van Dijkstra

Terminologie.

Een **graaf** is opgebouwd uit knopen (nodes, vertices) en lijnen (takken, edges) .

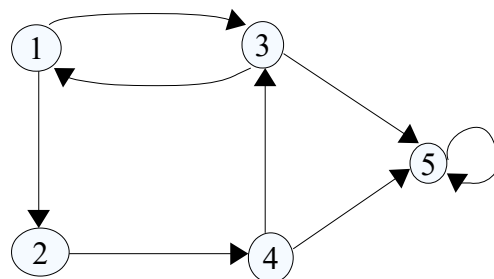
Een voorbeeld:



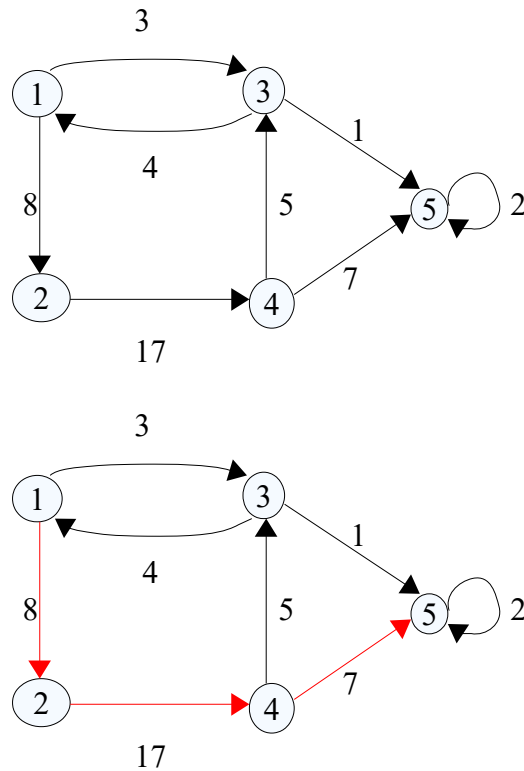
Een **lus** (loop) is een lijn die een punt met zichzelf verbindt

Een **enkelvoudige** graaf is een graaf zonder meervoudige lijnen.

Bij een **gerichte** graaf (directed graph) zijn de lijnen voorzien van pijlen.



Bij een **gewogen** graaf is aan een tak een getal (gewicht) toegevoegd.



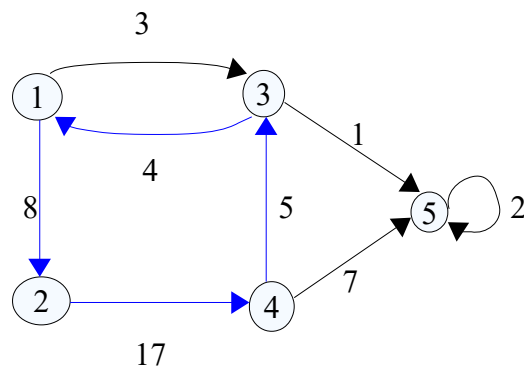
Pad (path): rij knopen die onderling verbonden zijn door een tak.

Een voorbeeld van een pad: $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$

De **ongewogen** padlengte is 3 (het aantal takken)

De **gewogen** padlengte is $8+17+7 = 32$ (de som van de gewichten)

Simpel pad: een pad waarbij een knoop 1 maal voorkomt.

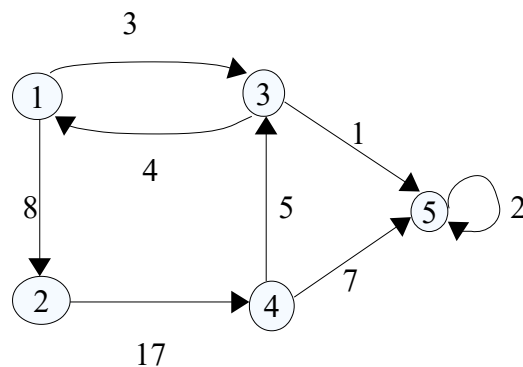


Een **cykel** is een pad waarbij de begin- en eindknoop gelijk zijn.

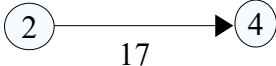
Een voorbeeld van een cykel: $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$

Wiskundige notaties.

Voorbeeld:



Een tak kan met de tuple (**<begin-node>** , **<eind-node>** , **gewicht**) worden vastgelegd.

De tak  wordt dus vastgelegd met de tuple **(2,4,17)**

De verzameling van alle nodes van de graaf is **$V = \{ 1,2,3,4,5 \}$**

De verzameling van alle takken van de graaf is **$E = \{ (1,3,3), (1,2,8), (2,4,17), (3,1,4), (3,5,1), (4,3,5), (4,5,7), (5,5,2) \}$**

De graaf wordt als volgt genoteerd:

$G = (V,E) = (\{ 1,2,3,4,5 \}, \{ (1,3,3), (1,2,8), (2,4,17), (3,1,4), (3,5,1), (4,3,5), (4,5,7), (5,5,2) \})$

('G' staat voor 'graph'. 'V' staat voor 'vertices'. 'E' staat voor 'edges')

De graad (degree) van een node is het aantal ingaande en uitgaande pijlen van de node.

In bovenstaand voorbeeld geldt:

$$\text{deg}(1) = 3$$

$$\text{deg}(2) = 2$$

$$\text{deg}(3) = 4$$

$$\text{deg}(4) = 3$$

$$\text{deg}(5) = 4 \text{ (een lus telt dubbel: het heeft een ingaande en een uitgaande pijl)}$$

$$\text{Er geldt } \text{deg}(1) + \text{deg}(2) + \dots + \text{deg}(5) = 3+2+4+3+4 = 16 = 2 * |E|$$

($|E|$: aantal pijlen)

$\text{indegree}(v)$: het aantal ingaande pijlen van v

$\text{outdegree}(v)$: het aantal uitgaande pijlen van v

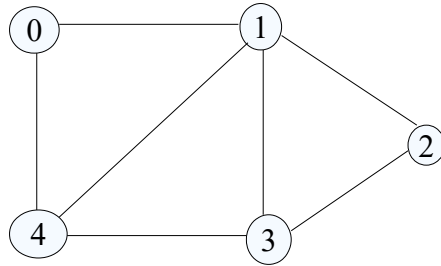
$$\text{Er geldt } \text{indeg}(1) + \text{indeg}(2) + \dots + \text{indeg}(5) = 1+1+2+1+3 = 8 = |E|$$

$$\text{outdeg}(1) + \text{outdeg}(2) + \dots + \text{outdeg}(5) = 2+1+2+2+1 = 8 = |E|$$

Representaties van een graaf.

Grafen kunnen op verschillende manieren gerepresenteerd worden.

Voorbeeld:



Representatie m.b.v. **adjacency-matrix**:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Representatie m.b.v. **adjacency-list**:

0	[1,4]
1	[0,2,3,4]
2	[1,3]
3	[1,2,4]
4	[0,1,3]

Een implementatie in Python.

(zie 25_Graph.py)

Voor een node wordt de volgende klasse gebruikt:

```
class Vertex:
    def __init__(self, data):
        self.data = data

    def __repr__(self):          # voor afdrukken
        return str(self.data)

    def __lt__(self, other):    # voor sorteren
        return self.data < other.data
```

De klasse 'Vertex' bevat alleen het data-attribuut.

Bij de algoritmen, die behandeld worden, worden aan een node extra attributen toegevoerd.

In Python kunnen attributen dynamisch toegevoegd en verwijderd worden.

De graaf wordt als volgt gedefinieerd:

```
v = [Vertex(i) for i in range(5)]

G = {v[0]:[v[1],v[4]],
      v[1]:[v[0],v[2],v[3],v[4]],
      v[2]:[v[1],v[3]],
      v[3]:[v[1],v[2],v[4]],
      v[4]:[v[0],v[1],v[3]]}
```

Het afdrukken van de nodes gaat als volgt:

```
def vertices(G):
    return sorted(G)

print("vertices(G):",vertices(G))
```

Uitvoer:

```
vertices(G): [0, 1, 2, 3, 4]
```

Het afdrukken van de takken gaat als volgt:

```
def edges(G):
    return [(u,v) for u in vertices(G) for v in G[u]]

print("edges(G):",edges(G))
```

Uitvoer:

```
edges(G): [(0, 1), (0, 4), (1, 0), (1, 2), (1, 3), (1, 4), (2, 1),
            (2, 3), (3, 1), (3, 2), (3, 4), (4, 0), (4, 1), (4, 3)]
```

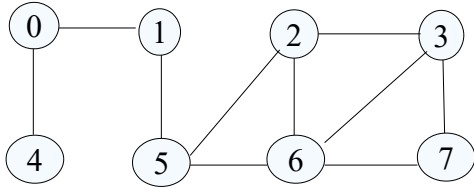
Het 'breadth first search' (BFS) algoritme.

Ga uit van een graaf G en een node s.

Ga na welke nodes bereikbaar zijn vanuit s. Bepaal bij iedere node de afstand (d.w.z. het kleinste aantal takken) van s tot die node. Geef ook aan welk pad daarvoor moet worden bewandeld.

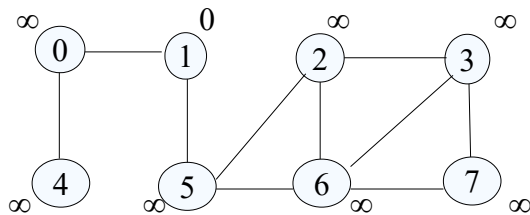
Het algoritme wordt behandeld aan de hand van een voorbeeld.

Graaf G:

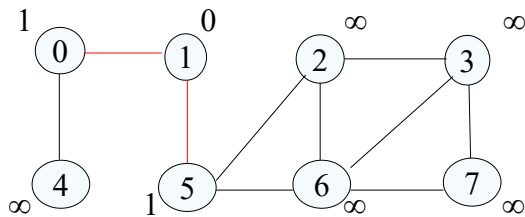


Ga uit van node 1.

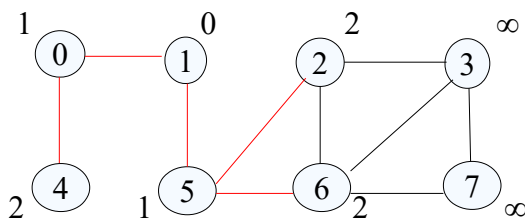
Stap 1: Node '1' krijgt afstand '0'. Alle andere nodes krijgen afstand ' ∞ '.



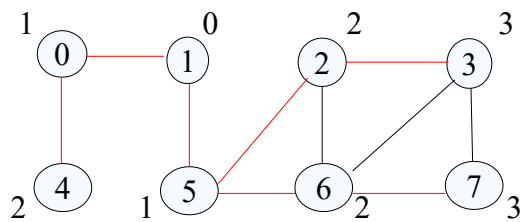
Stap 2: De burenen van node '1' krijgen afstand '1'.



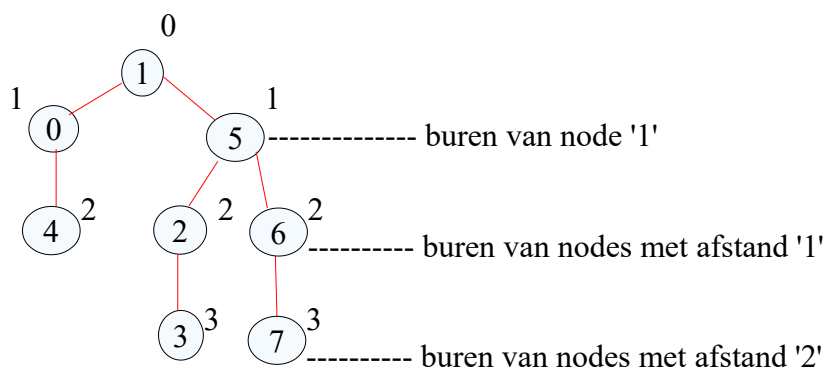
Stap 3: De burenen van de nodes met afstand '1' krijgen afstand '2'.



Stap 4: De buren van de nodes met afstand '2' krijgen afstand '3'.



Als de zwarte takken worden weggelaten kan de graaf als volgt worden weergegeven:



Deze (sub)graaf bevat geen cycli. Zo'n graaf wordt een **tree** genoemd.

Omdat de tree alle nodes van de oorspronkelijke graaf bevat het een **spanning tree**.

Bij een boom heeft, behalve de root, iedere node één voorganger (predecessor).

De boom kan ook vastgelegd worden door van iedere node aan te geven welke node de voorganger is.

	0	1	2	3	4	5	6	7
predecessor:	1	–	5	2	0	1	5	6

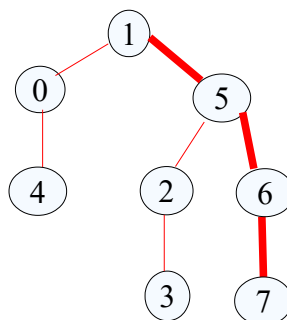
Het pad van node 0 naar node 7 wordt nu als volgt bepaald:

$\text{predecessor}[7] = 6$

$\text{predecessor}[6] = 5$

$\text{predecessor}[5] = 1$

Het pad is dus $1 \rightarrow 5 \rightarrow 6 \rightarrow 7$



De implementatie van BFS.

(zie 26_BFS.py)

Ga uit van een graaf G en een node s.

Ga na welke nodes bereikbaar zijn vanuit s. Bepaal bij iedere node de afstand (het kleinste aantal takken) van s tot die node. Geef ook aan welk pad daarvoor moet worden bewandeld.

Bij de implementatie wordt gebruik gemaakt van een queue.

Eerst wordt de startnode in de queue gezet.

Vervolgens worden de burens van de startnode in de queue gezet.

Het begrip "oneindig" wordt als volgt geïmplementeerd;

```
import math
INFINITY = math.inf
```

De graaf G wordt als volgt vastgelegd:

```
v = [Vertex(i) for i in range(8)]

G = {v[0]:[v[1],v[4]],
      v[1]:[v[0],v[5]],
      v[2]:[v[3],v[5],v[6]],
      v[3]:[v[2],v[6],v[7]],
      v[4]:[v[0]],
      v[5]:[v[1],v[2],v[6]],
      v[6]:[v[2],v[3],v[5],v[7]],
      v[7]:[v[3],v[6]]}
```

Het doorlopen van de graaf gaat als volgt:

```
def BFS(G,s):
    V = vertices(G)
    s.predecessor = None # s krijgt het attribuut 'predecessor'
    s.distance = 0      # s krijgt het attribuut 'distance'
    for v in V:
        if v != s:
            v.distance = INFINITY # v krijgt het attribuut 'distance'
    q = myqueue()
    q.enqueue(s) # plaats de startnode in de queue
    while q:
        u = q.dequeue()
        for v in G[u]:
            if v.distance == INFINITY: # v is nog niet bezocht
                v.distance = u.distance + 1
                v.predecessor = u # v krijgt het attribuut 'predecessor'
                q.enqueue(v)      # plaats v in de queue

BFS(G,v[1])
```

De opgebouwde boom kan nu als volgt getoond worden:

```
def show_tree_info(G):
    print('tree:', end = ' ')
    for v in vertices(G):
        print('(' + str(v), end = '')
        if hasattr(v, 'distance'):
            print(',d:' + str(v.distance), end = '')
        if hasattr(v, 'predecessor'):
            print(',p:' + str(v.predecessor), end = '')
        print(')', end = ' ')
    print()

show_tree_info(G)
```

Uitvoer:

```
tree: (0,d:1,p:1) (1,d:0,p:None) (2,d:2,p:5) (3,d:3,p:2) (4,d:2,p:0)
      (5,d:1,p:1) (6,d:2,p:5) (7,d:3,p:6)
```

De nodes worden als volgt gesorteerd op basis van 'distance' en vervolgens op 'predecessor':

```
def show_sorted_tree_info(G):
    print('sorted tree:')
    V = vertices(G)
    V = [v for v in V if hasattr(v, 'distance') and
                                                hasattr(v, 'predecessor')]
    V.sort(key = lambda x: (x.distance,x.predecessor))
    d = 0
    for v in V:
        if v.distance > d:
            print()
            d += 1
            print('(' + str(v) + ',d:' + str(v.distance) + ',p:'
                  + str(v.predecessor), end = '')
            print(')', end = ' ')
    print()

show_sorted_tree_info(G)
```

Uitvoer:

```
sorted tree:
(1,d:0,p:None)
(0,d:1,p:1) (5,d:1,p:1)
(4,d:2,p:0) (2,d:2,p:5) (6,d:2,p:5)
      (3,d:3,p:2) (7,d:3,p:6)
```

Het pad van $v[0]$ naar $v[7]$ wordt als volgt verkregen:

```
def path_BFS(G,u,v):
    BFS(G,u)
    a = []
    if hasattr(v, 'predecessor'):
        current = v
        while current:
            a.append(current)
            current = current.predecessor
        a.reverse()
    return a

print("path_BFS(G,v[1],v[7]):",path_BFS(G,v[1],v[7]))
```

Uitvoer:

```
path_BFS(G,v[1],v[7]): [1, 5, 6, 7]
```

De nodes van de graaf zijn nu voorzien voor de attributen 'distance' en 'predecessor' .

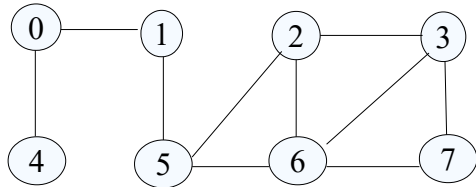
De graaf kan als volgt worden schoongemaakt:

```
def clear(G): # verwijder alle toegevoegde attributen van de nodes
    for v in vertices(G):
        k = [e for e in vars(v) if e != 'data']
        for e in k:
            delattr(v,e)
```

Het 'depth first search' (DFS) algoritme.

Bij 'depth first search' gaat het zoeken zo diep mogelijk de graaf in. Als een eindpunt bereikt is, wordt een stap terug gedaan.

We gaan uit van dezelfde graaf.

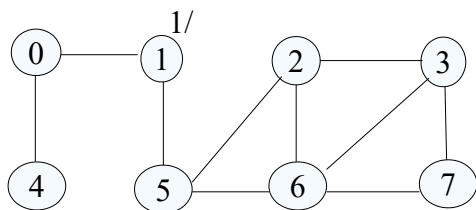


Ga uit van node '1'.

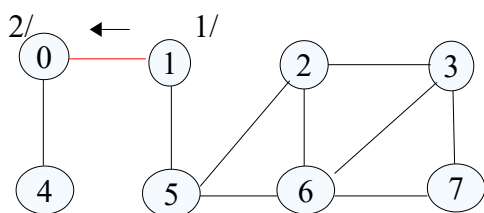
Er wordt een teller bijgehouden.

Zodra een node wordt ontdekt wordt de teller opgehoogd en wordt de waarde van de teller bij de node geplaatst. Nadat de nakomelingen van de node zijn bezocht wordt de teller weer opgehoogd. Ook deze waarde wordt bij de node geplaatst.

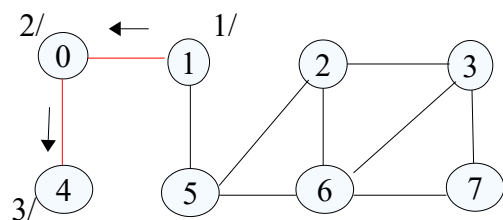
Stap 1:



Stap 2: Node '0' wordt ontdekt.

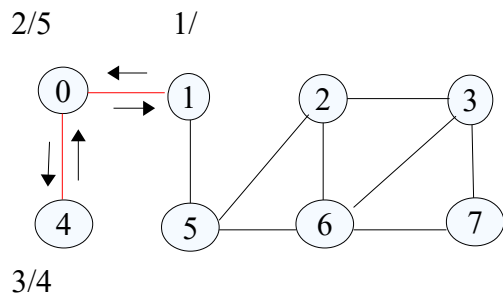


Stap 3: Node '4' wordt ontdekt.

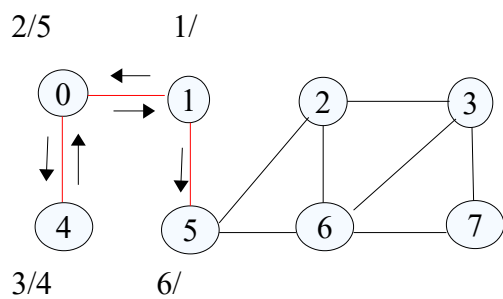


Stap 4: Vanaf node '4' kan niet dieper de boom in gezocht worden.

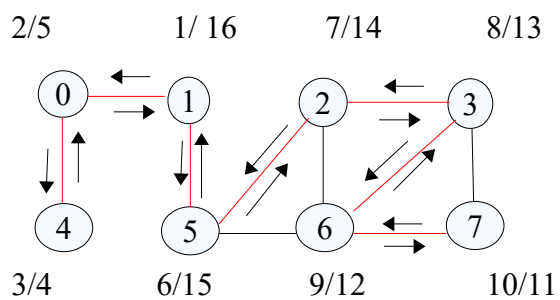
Keer terug naar node '0'. Keer vervolgens terug naar node '0' .



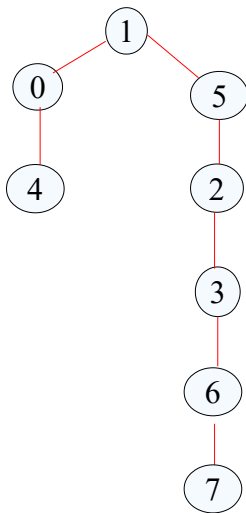
Stap 5: Node '5' wordt ontdekt.



Na een aantal stappen:



Als de zwarte takken worden weggelaten kan de graaf als volgt worden weergegeven:



De implementatie van DFS.

(zie 27_DFS.py)

Bij de implementatie van BFS werd gebruik gemaakt van een queue.

Bij de implementatie van DFS wordt gebruik gemaakt van recursie.

Het had ook opgelost kunnen worden m.b.v. een stack: recursie maakt impliciet gebruik van een stack.

Het doorlopen van de graaf gaat als volgt:

```
def DFS(G,s):
    global time
    time += 1
    s.discover = time
    for v in G[s]:
        if not (hasattr(v, 'discover')):
            v.predecessor = s
            DFS(G,v)
    time += 1
    s.finish = time

time = 0
v[1].predecessor = None
DFS(G,v[1])
```

Aan een node worden drie attributen toegevoegd: discover, predecessor en finish. De attributen 'discover' en 'finish' zijn alleen nodig voor het monitoren van het doorloop-proces.

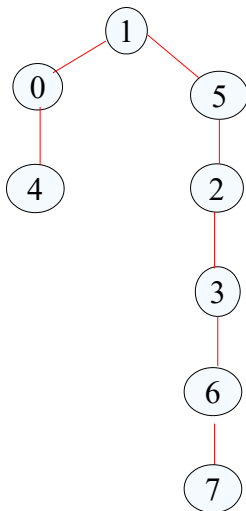
De opgebouwde boom wordt als volgt getoond:

```
def show_tree_info(G):
    print('tree:', end = ' ')
    for v in vertices(G):
        print('(' + str(v), end = ' ')
        if hasattr(v, 'discover'):
            print(',d:' + str(v.discover), end = ' ')
        if hasattr(v, 'finish'):
            print(',f:' + str(v.finish), end = ' ')
        if hasattr(v, 'predecessor'):
            print(',p:' + str(v.predecessor), end = ' ')
        print(')', end = ' ')
    print()

show_tree_info(G)
```

Uitvoer:

```
tree: (0,d:2,f:5,p:1) (1,d:1,f:16,p:None) (2,d:7,f:14,p:5)
      (3,d:8,f:13,p:2) (4,d:3,f:4,p:0)      (5,d:6,f:15,p:1)
      (6,d:9,f:12,p:3) (7,d:10,f:11,p:6)
```



De minimale opspanningsboom.

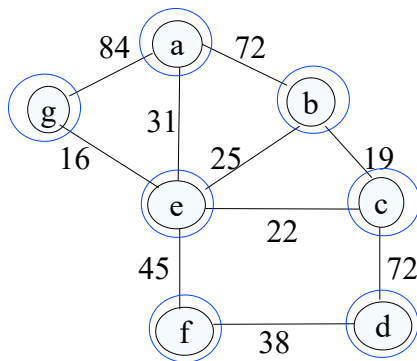
Hoe bepalen we bij een gegeven graaf de opspanningsboom met de kortste lengte?

Een dergelijke boom heet een 'minimal spanning tree' (MST).

Hier bestaan twee algoritmen voor: van Joseph Kruskal en van Robert Prim.

Het MST-algoritme van Kruskal.

Ga uit van het volgende voorbeeld:



Tijdens het algoritme worden twee dingen bijgehouden:

- een lijst van takken, lijnen gesorteerd naar gewicht:

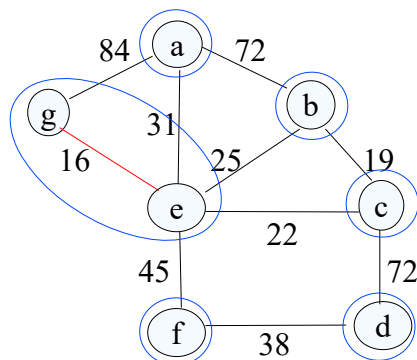
eg : 16 , bc : 19 , ce : 22 , be : 25 , ae : 31 , df : 38 , ef : 45 , ab : 72 , cd : 72 , ag : 84

- Een partitie van de verzameling van nodes.
(Een partitie is een opsplitsing van het geheel in een set deelverzamelingen.
De deelverzamelingen zijn onderling disjunct, d.w.z. ze hebben geen gemeenschappelijke elementen.)

Begonnen wordt met de partitie **{a} , {b} , {c} , {d} , {e} , {f} , {g}**
(alle nodes staan los van elkaar)

Stap 1: maak de kleinste tak (i.c. 'eg') uit de lijst rood en verwijder de tak uit de lijst

Lijst: **bc : 19 , ce : 22 , be : 25 , ae : 31 , df : 38 , ef : 45 , ab : 72 , cd : 72 , ag : 84**

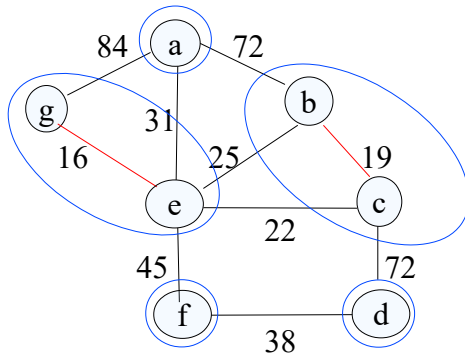


Nodes, die tot dezelfde rode subtree behoren worden samengevoegd in één set.

De partitie wordt: **{a} , {b} , {c} , {d} , {e,g} , {f}**

Stap 2: maak kleinste tak (i.c. 'bc') uit de lijst rood en verwijder tak uit de lijst

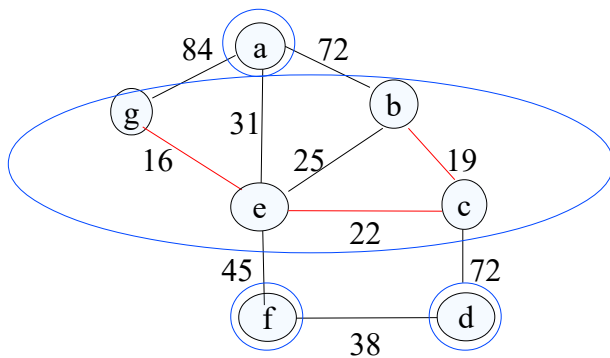
Lijst: **ce : 22 , be : 25 , ae : 31 , df : 38 , ef : 45 , ab : 72 , cd : 72 , ag : 84**



Partitie: **{a} , {b,c} , {d} , {e,g} , {f}**

Stap 3: maak de kleinste tak (i.c. 'ce') uit de lijst rood en verwijder de tak uit de lijst

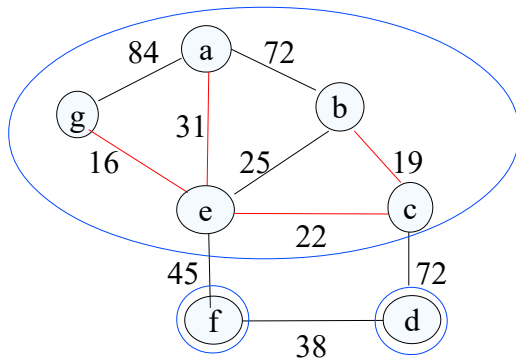
Lijst: **be : 25 , ae : 31 , df : 38 , ef : 45 , ab : 72 , cd : 72 , ag : 84**



Partitie: **{a} , {b,c,e,g} , {d} , {f}**

Stap 4: de tak 'be' kan niet toegevoegd worden, omdat anders een cykel ontstaat.
De volgende tak is 'ae'

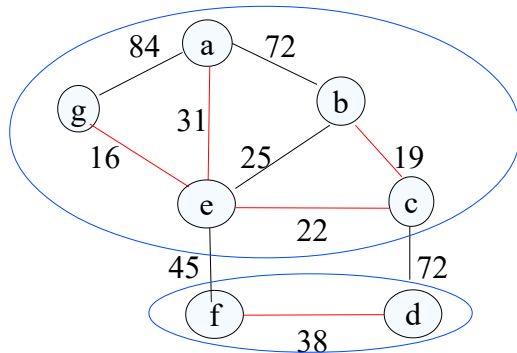
Lijst: **df : 38 , ef : 45 , ab : 72 , cd : 72 , ag : 84**



Partitie: **{a,b,c,e,g} , {d} , {f}**

Stap 5: maak de kleinste tak (i.c. 'ce') uit de lijst rood en verwijder de tak uit de lijst

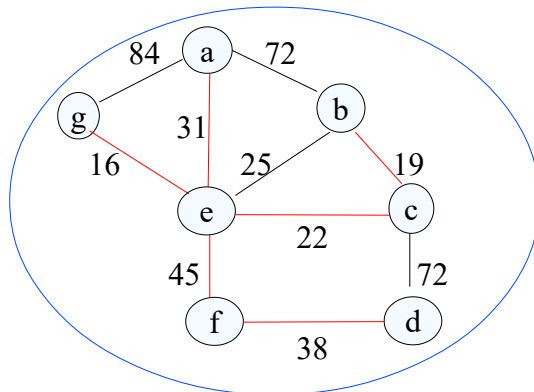
Lijst: **ef : 45** , **ab : 72** , **cd : 72** , **ag : 84**



Partitie: **{a,b,c,e,g}** , **{d,f}** (er zijn twee rode subtrees)

Stap 6: maak de kleinste tak (i.c. 'ef') uit de lijst rood en verwijder de tak uit de lijst

Lijst: **ab : 72** , **cd : 72** , **ag : 84**



Partitie: **{a,b,c,d,e,f,g}**

De partitie bestaat uit één verzameling en de overige takken kunnen niet worden toegevoegd.
Het proces is afgerond.

De partitie wordt bijgehouden om na te gaan of een tak wel toegevoegd mag worden.

De implementatie van het Kruskal-algoritme.

De graaf wordt als volgt geïmplementeerd:

```
v = [Vertex(i) for i in list('abcdefg')]

G = {v[0]:{v[1]:72,v[4]:31,v[6]:84},
      v[1]:{v[0]:72,v[2]:19,v[4]:25},
      v[2]:{v[1]:19,v[3]:72,v[4]:22},
      v[3]:{v[2]:72,v[5]:38},
      v[4]:{v[0]:31,v[1]:25,v[2]:22,v[5]:45,v[6]:16},
      v[5]:{v[3]:38,v[4]:45},
      v[6]:{v[0]:84,v[4]:16}}
```

Het sorteren van de takken gebeurt als volgt:

```
def sorted_edges(G):
    return sorted(edges(G),key=lambda x: x[2])
                                # sorteer naar gewicht
```

De minimale opspanningsboom wordt als volgt verkregen:

```
def MST_Kruskal(G):
    A = []
    V = vertices(G)
    find_set = {}
    for v in V:
        find_set[v] = set([v]) # partitie: v is element van
                                # find_set[v]
    E = sorted_edges(G)        # gesorteerd op basis van gewicht
    for (u,v,w) in E:
        if find_set[u] != find_set[v]:
            # u en v behoren niet tot dezelfde set
            A.append((u,v,w))    # heen
            A.append((v,u,w))    # terug
            s = find_set[u] | find_set[v]
            # de vereniging van find_set[u] en find_set[v]
            for e in s:
                find_set[e] = s    # pas partitie aan
    return A

print("MST_Kruskal(G):",MST_Kruskal(G))
```

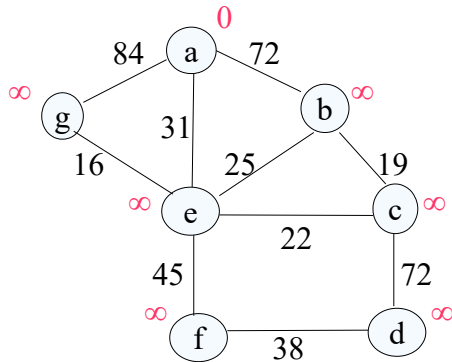
Uitvoer:

```
MST_Kruskal(G): [(e, g, 16), (g, e, 16), (b, c, 19), (c, b, 19),
                  (c, e, 22), (e, c, 22), (a, e, 31), (e, a, 31),
                  (d, f, 38), (f, d, 38), (e, f, 45), (f, e, 45)]
```

Het MST-algoritme van Prim.

Bij het algoritme van Kruskal werd uitgegaan van partities.
Het algoritme van Prim gaat uit van een startnode. Van daaruit wordt de minimale opspanningsboom opgebouwd

Ga uit van hetzelfde voorbeeld:



Neem als startpunt 'a' .

De node 'a' krijgt label '0'. De andere nodes krijgen label ' ∞ ' .

Tijdens het algoritme worden een boom opgebouwd. Iedere node wordt voorzien van twee labels:

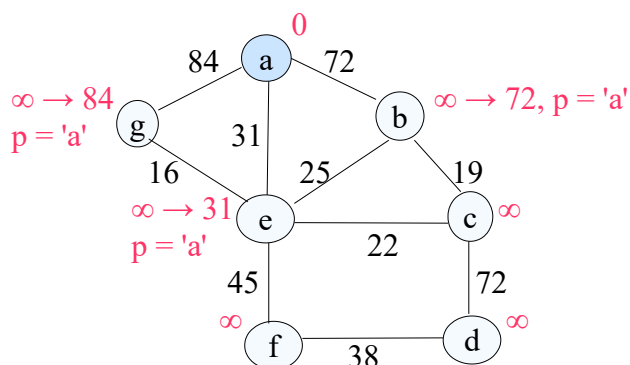
- het gewicht van de tak tussen de voorganger van de node en de node
- de voorganger (predecessor, p)

Er wordt een lijst Q van nodes bijgehouden.

Q bevat eerst alle nodes, gesorteerd naar label:

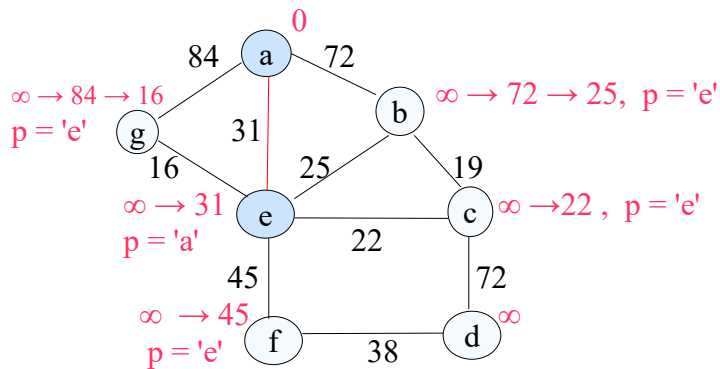
Q: **a:0** , b: ∞ , c: ∞ , d: ∞ , e: ∞ , f: ∞ , g: ∞

Stap 1: verwijder node (i.c. 'a') met kleinste label uit Q. Als het gewicht van de tak tussen de verwijderde node en een buur kleiner is dan de label-waarde van de buur, wordt de label-waarde aangepast. Ook wordt de 'predecessor' aangepast. Sorteer Q opnieuw naar label.



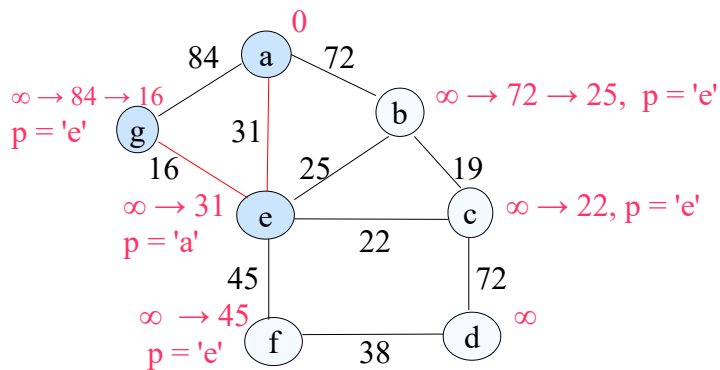
Q: **e:31** , **b:72** , **g:84** , c: ∞ , d: ∞ , f: ∞

Stap 2: verwijder node 'e' en pas de labels van 'b', 'c', 'f' en 'g' aan.



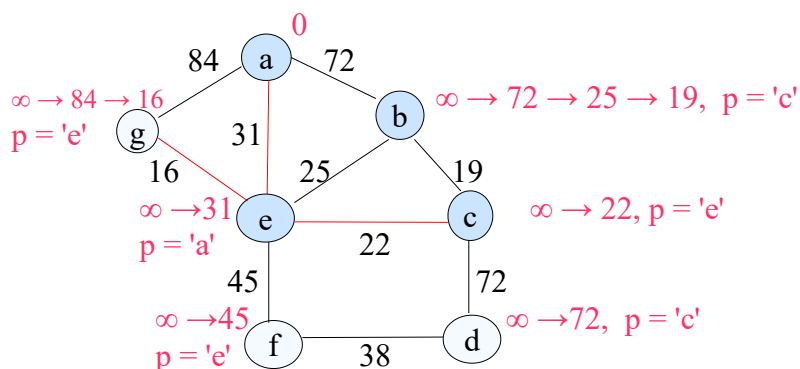
Q: **g:16 , c:22 , b:25 , f: 45 , d:∞**

Stap 3: verwijder node 'g'. Er hoeven geen labels aangepast te worden.



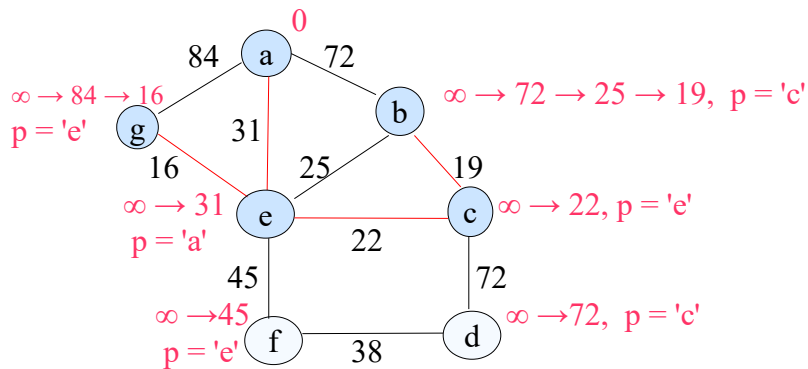
Q: **c:22 , b:25 , f: 45 , d:∞**

Stap 4: verwijder node 'c' en pas de labels 'b' en 'd' aan.



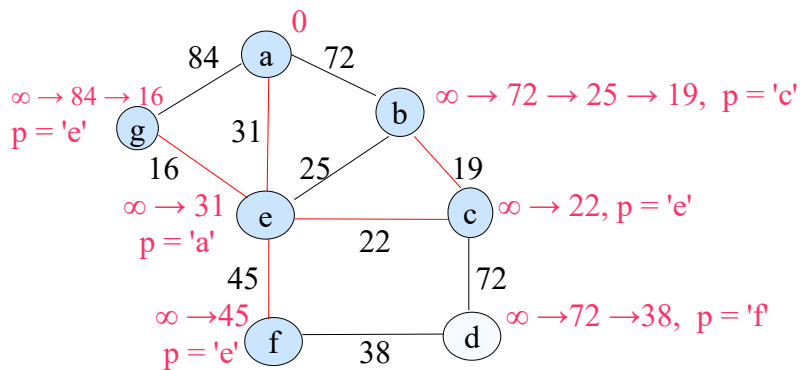
Q: **b:19 , f: 45 , d:72**

Stap 5: verwijder node 'b' . Er hoeven geen labels aangepast te worden.



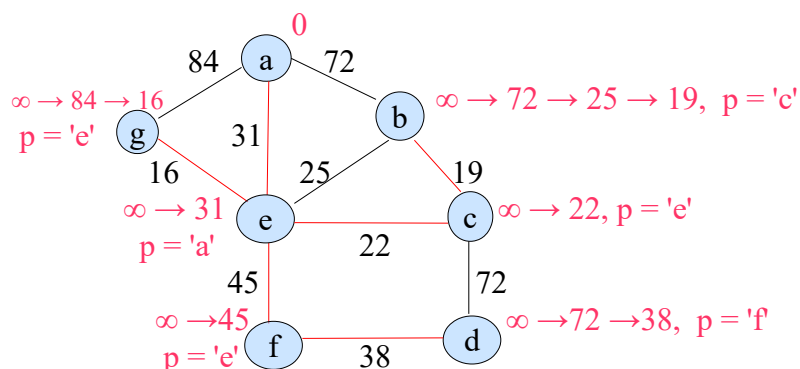
Q: **f:45, d:72**

Stap 6: verwijder node 'f' en pas label van 'd' aan.



Q: **d:38**

Stap 7: verwijder node 'd' .



De implementatie van het Prim-algoritme.

De minimale opspanningsboom wordt als volgt verkregen:

```
def MST_Prim(G,s):
    Q = vertices(G)
    for v in Q:
        v.key = INFINITY
        v.predecessor = None
    s.key = 0
    while Q:
        u = min(Q,key=lambda x: x.key)
        Q.remove(u)
        for v in G[u].keys():
            if v in Q and G[u][v] < v.key:
                v.predecessor = u
                v.key = G[u][v]

MST_Prim(G,v[0])

def tree_edges(G):
    a = []
    for v in vertices(G):
        if hasattr(v,'predecessor') and v.predecessor != None:
            a.append((v,v.predecessor,G[v][v.predecessor]))
            a.append((v.predecessor,v,G[v.predecessor][v]))
    a.sort()
    return a

print('tree_edges(G):',tree_edges(G))

print([(v,v.key) for v in vertices(G)])
print(sum([v.key for v in vertices(G)]))
```

Uitvoer:

```
tree_edges(G): [(a, e, 31), (b, c, 19), (c, b, 19), (c, e, 22),
               (d, f, 38), (e, a, 31), (e, c, 22), (e, f, 45),
               (e, g, 16), (f, d, 38), (f, e, 45), (g, e, 16)]
```

```
[(a, 0), (b, 19), (c, 22), (d, 38), (e, 31), (f, 45), (g, 16)]
```

171

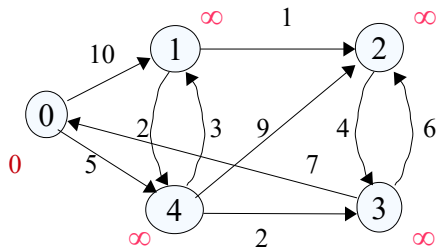
Het 'korste pad' bepalen.

Hoe bepalen we bij een gerichte graaf het kortste pad tussen twee gegeven nodes?

Dit wordt berekend m.b.v. het Dijkstra-algoritme.

Het algoritme van Dijkstra.

Ga uit van het volgende voorbeeld:



Neem als startpunt '0' .

De node '0' krijgt label '0'. De andere nodes krijgen label '∞' .

Tijdens het algoritme worden een boom opgebouwd. Iedere node wordt voorzien van twee labels:

- de lengte van het pad van '0' naar de node
- de voorganger (predecessor, p)

Er wordt een lijst Q van nodes bijgehouden.

Q bevat eerst alle nodes, gesorteerd naar label:

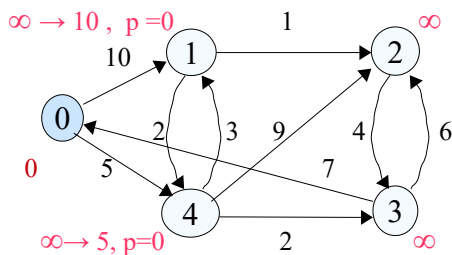
Q: **0:0** , 1:∞ , 2:∞ , 3:∞ , 4:∞

Stap 1: verwijder node (i.c. '0') met kleinste label uit Q.

Als een node u wordt verwijderd en bij een buur v geldt: $v.\text{label} > u.\text{label} + \text{'lengte tak (u,v)'}'$ vervang v.label door $u.\text{label} + \text{'lengte tak (u,v)'}'$.

Pas ook de 'predecessor' aan.

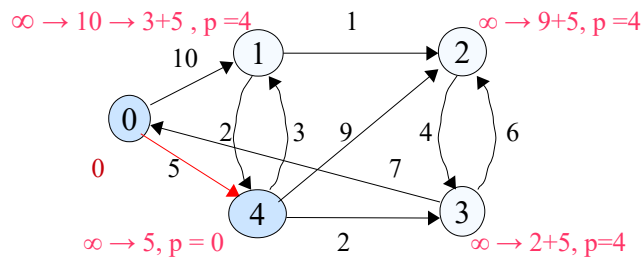
Sorteer Q opnieuw naar label.



Pas de nodes '1' en '4' aan.

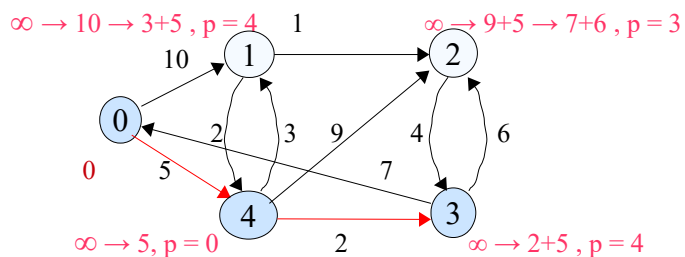
Q: **4:5** , 1:10 , 2:∞ , 3:∞

Stap 2: verwijder node '4' uit Q.
 Pas de nodes '1', '2' en '3' aan.
 Sorteer Q opnieuw naar label.



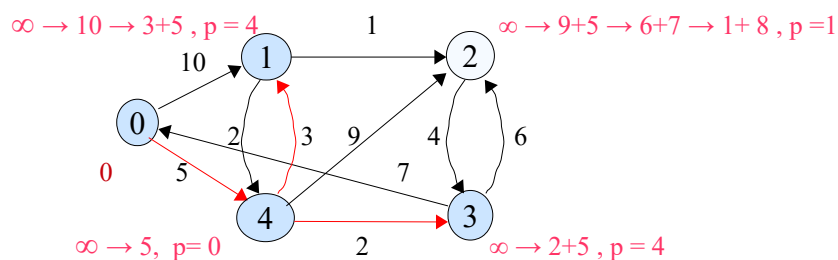
Q: **3:7, 1:8, 2:14**

Stap 3: verwijder node '3' uit Q.
 Pas de node '2' aan.
 Sorteer Q opnieuw naar label.



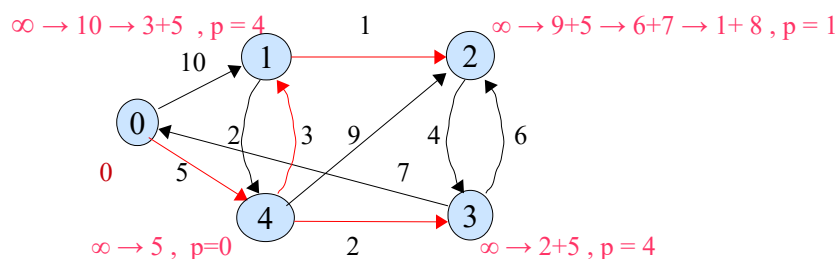
Q: **1:8, 2:13**

Stap 3: verwijder node '1' uit Q.
 Pas de node '2' aan.



Q: **2:9**

Stap 4: verwijder node '2' uit Q.



De implementatie van het Dijkstra-algoritme.

```
def Dijkstra(G,s):
    Q = vertices(G)
    for v in V:
        v.d = INFINITY
        v.predecessor = None
    s.d = 0

    while Q:
        u = min(Q,key=lambda x: x.d)
            # u is het element uit Q met de kleinste distance
        Q.remove(u)
        for v in G[u].keys():
            # v is een buur van u
            if v in Q and v.d > G[u][v]+ u.d:
                # pas indien nodig de distance van v aan
                v.predecessor = u
                v.d = G[u][v] + u.d

Dijkstra(G,v[0])

def tree_edges(G):
    a = []
    for v in vertices(G):
        if hasattr(v, 'predecessor') and v.predecessor != None:
            a.append((v.predecessor,v,G[v.predecessor][v]))
    a.sort()
    return a

print('tree_edges(G):',tree_edges(G))

print([(v,v.d) for v in vertices(G)])

def shortest_path(G,u,v):
    clear(G)
    Dijkstra(G,u)
    a = []
    if hasattr(v, 'predecessor'):
        current = v
        while current:
            a.append(current)
            current = current.predecessor
    a.reverse()
    return a

a = shortest_path(G,v[0],v[2])
print(a)
```

Uitvoer:

```
tree_edges(G): [(0, 4, 5), (1, 2, 1), (4, 1, 3), (4, 3, 2)]
[(0, 0), (1, 8), (2, 9), (3, 7), (4, 5)]
[0, 4, 1, 2]
```

De performance van algoritmen.

BFS:

Alle edges worden m.b.v. een queue opgezocht: $O(|E|)$

DFS:

Alle edges worden recursief opgezocht: $O(|E|)$

Algoritme van Kruskal:

Het sorteren van edges is het meeste werk: $O(|E| \ln(|E|)) = O(|E| \ln(|V|))$

Algoritme van Prim:

De performance kan worden geoptimaliseerd door gebruik te maken van een priority-queue.
Als de priority-queue wordt gerealiseerd met 'Fibonacci Heap', is de performance:

$O(|E| + |V| \ln |V|)$

Algoritme van Dijkstra

Net als bij het algoritme van Prim: $O(|E| + |V| \ln |V|)$