Inleiding Programmeren
College 12

Universiteit van Amsterdam

Robin de Vries

16 oktober 2017

Inhoud

Sterkte van operatoren

Bij expressies zonder haakjes is het goed om te weten welke operatoren voorrang hebben op anderen:

Operators	**Presedence**
postfix	`expr++ expr--`
unary	`++expr --expr +expr -expr ~ !`
multiplicative	`* / %`
additive	`+ -`
shift	`<< >> >>>`
relational	`> > <= >= instanceof`
equality	`== !=`
bitwise AND	`&`
bitwise exclusive OR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
logical OR	`||`
ternary	`? :`
assignment	`= += -= *= /= &= ^= |= <<= >>= >>>=`

Sterkte van operatoren

Plaats de haakjes in onderstaande expressie:

e = a * b + c == −a / b && ! d

Antwoord:

$$e = \Big(\Big(\big((a * b) + c\big) == \big((-a)/b\big)\Big) \&\& (!d)\Big)$$

Inhoud

Datastructuren

Veel programma's en algoritmen hebben een complexere manier nodig om gegevens in op te slaan dan in losse variabelen.
Gegevens worden dan opgeslagen in een: **datastructuur**.

Er zijn velen datastructuren al zeer uitgebreid beschreven, waaronder:

- Stack
- Heap
- Lijsten
- Hashtables
- Bomen
- Grafen

In periode 4 volgt voor de Informatici het vak **Datastructuren** waarin al deze datastructuren (en veel meer) worden behandeld.

Datastructuren (II)

Uiteraard zitten in Java ook al vele datastructuren ingebouwd.
Om te zorgen dat deze datastructuren eenvoudig inwisselbaar zijn heeft Java:
Abstract Data Types (ADTs).

Een abstract datatype is een interface dat vertelt waar een bepaalde datastructuur aan moet voldoen.

Een voorbeeld hiervan is `java.util.List<E>`[1] dat definieert dat een lijst `add(E)`, `get(int)` en `size()` methoden moet hebben. Een `ArrayList` is dan vervolgens een implementatie van een `List`.

[1] `http://docs.oracle.com/javase/8/docs/api/java/util/List.html`

Inhoud

HashMap

Een HashMap is een implementatie van een **java.util.map**[2].
Het idee van een map is dat het **sleutel**s aan **waarde**n toekent. Bijvoorbeeld:

```
Map<String, Integer> lijstje = new HashMap<String, Integer>();
lijstje.put("Een", 1);
lijstje.put("Twee", 2);

System.out.println(lijstje.get("Twee"));
```

```
2
```

[2] https://docs.oracle.com/javase/7/docs/api/java/util/Map.html

HashMap (2)

Het is bij een hashmap ongebruikelijk dat je de sleutels kunt achterhalen, je kunt dus achteraf **niet** opvragen welke sleutels erin zijn gestopt. Je kunt wel over alle waarden heen itereren:

```
Map<String, Integer> lijstje = new HashMap<String, Integer>();
lijstje.put("Een", 1);
lijstje.put("Twee", 2)
for(Integer value : lijstje.values())
{
    System.out.println(value);
}
```

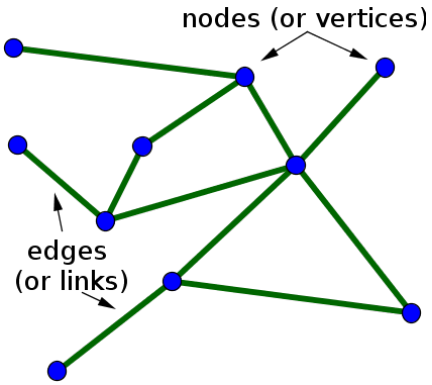
```
2
1
```

Je hebt echter niet altijd garantie over de volgorde waarin objecten zijn opgeslagen!

Inhoud

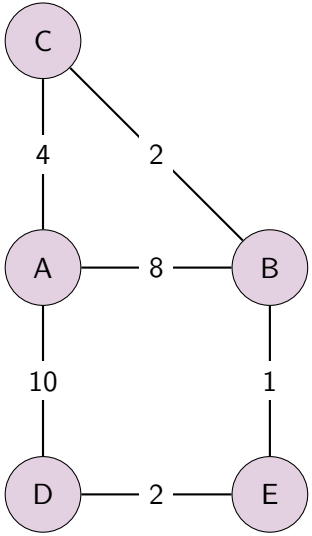
Grafen

- Een graaf is een verzameling punten (ook wel knopen genoemd).
- Deze knopen zijn verbonden door lijnen (zijden, kanten).
- Een verzameling kanten om van knoop A naar B te komen, wordt een pad genoemd.
- Wij beschouwen op dit moment alleen **ongerichte** grafen.



Grafen en paden

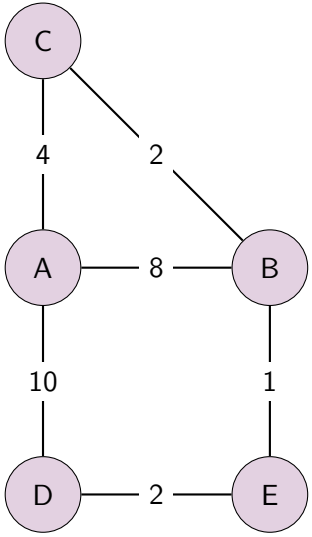
- De kanten van een graaf kunnen ook gewichten hebben.
- A-B is een pad van A naar B, maar A-D-E-B is dat ook.
- Van A naar B via A-B kost dus 8.
- van A naar B via A-D-E-B kost dus $(10 + 2 + 1 =)$ 13.
- Het pad dat je van het ene punt naar het andere punt brengt, zonder dat er een ander pad is met een lager gewicht, heet het: **kortste pad**.



Eigenschappen van grafen

Naast paden hebben we ook nog diverse andere eigenschappen van grafen en knopen:

- Afstand tussen twee knopen: lengte van het kortste pad
- Diameter: maximale lengte van het kortste pad
- Graad van een knoop: aantal kanten dat de knoop verbindt met andere knopen (in eenvoudige gevallen)



Inhoud

Dijkstra's korste pad algoritme

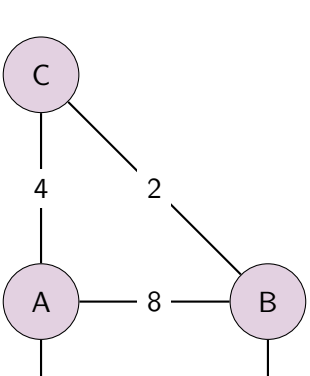
Een eenvoudig algoritme om het kortste pad tussen twee knopen te bepalen werd in 1959 door de Nederlandse informaticus Edsger Dijkstra beschreven.[3]

Dit algoritme maakt gebruik van de (vanzelfsprekende) aanname dat als een punt R op het kortste pad van P naar Q ligt, dan ook dat pad van P naar R het kortste pad is.

Het algoritme doorloopt de graaf en bekijkt daarvoor verschillende paden in toenemende grootte, zodat het uiteindelijk voor één knoop alle kortste paden kent.

[3] Edsger W Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), p. 269–271.

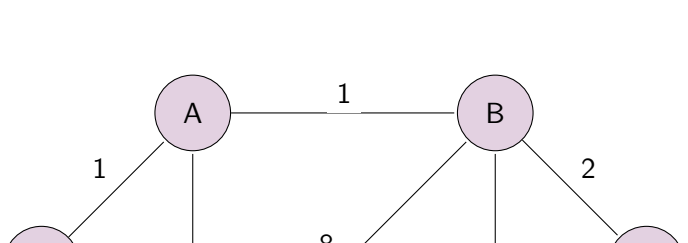
Voorbeeld (I)



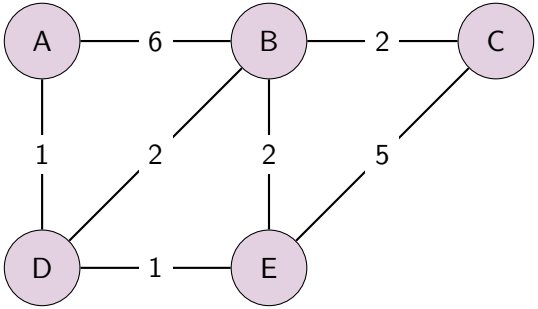
Kortste paden vanaf A:

x	A	B	C	D	E
A	$\boxed{0_A}$	8_A	4_A	10_A	∞
C		6_C	$\boxed{4_A}$	10_A	∞
B		$\boxed{6_C}$		10_A	7_B
E				9_E	$\boxed{7_B}$
D				$\boxed{9_E}$	

Voorbeeld (II)



Voorbeeld (III)



Inhoud

Grafen in Java

Leuk zo'n graaf, maar hoe representeer je dat nu met een computer?

Of specifieker: OOP in Java?

Grafen in Java

We gaan gebruikmaken van inner klassen!

```
public class Graph {
    HashMap<String, Vertex> vertices = new HashMap<>();
    ArrayList<Edge> edges = new ArrayList<>();
    class Edge {
        Vertex point1, point2;
        int weight;
        Edge(Vertex p1, Vertex p2, int w) {
            point1=p1; point2=p2; weight=w;
        }
    }
    class Vertex {
        String name;
        ArrayList<Edge> edges = new ArrayList<>();
        Vertex(String name) { this.name = name;}
        public String toString() { return name; }
    }
}
```

Toevoegen van een knoop

Omdat `Vertex` een inner klasse is, kunnen we hem alleen binnen een instantie van een Graph instantieren.

```
class Graph {
    HashMap<String, Vertex> vertices = new HashMap<>();
    // code van net

    public void addVertex(String name) {
        vertices.put(name, new Vertex(name));
    }

    // code van net
    class Edge { }
    class Vertex { }
}
```

Toevoegen van een kant

```
class Graph {

    // code van net

    public void addEdge(String name1, String name2, int weight) {
        Vertex v1 = vertices.get(name1);
        Vertex v2 = vertices.get(name2);
        Edge newEdge = new Edge(v1, v2, weight);
        v1.edges.add(newEdge);
        v2.edges.add(newEdge);
        edges.add(newEdge);
    }

    // code van net
    class Edge { }
    class Vertex { }
}
```

Graaf maken

Nu we de klassen in basale vorm hebben, kunnen we een simpele graaf definiëren:

```
Graph graph = new Graph();
graph.addVertex("V1");
graph.addVertex("V2");
graph.addVertex("V3");
graph.addEdge("V1", "V2", 10);
graph.addEdge("V2", "V3", 20);

Graph.Vertex v2 = graph.vertices.get("V2");
System.out.println("Kanten van V2:");
for(Graph.Edge e : v2.edges) {
    System.out.printf("Kant %s-%s, gewicht: %d\n",
         e.point1, e.point2, e.weight);
}
```

```
Kanten van V2:
Kant V1–V2, gewicht: 10
Kant V2–V3, gewicht: 20
```

Inhoud

Anonieme functies

In veel gevallen is het handig om een functie mee te geven als argument aan een andere functie.

In veel andere programmeertalen (C#, Python, PHP, ..) zit dit al een tijdje, maar sinds kort ook in Java!

Appeltjes

Aanschouw de volgende klasse:

```
class Appel {
    String kleur;
    int gewicht;

    Appel(String kleur, int gewicht) {
        this.kleur = kleur;
        this.gewicht = gewicht;
    }

    public String toString() {
        return String.format("%s: %d gram", kleur, gewicht);
    }
}
```

Lijst van appeltjes

Stel dat we een verzameling van appels hebben en sommige willen filteren:

```
ArrayList<Appel> appeltjes = new ArrayList<>();

appeltjes.addAll(Arrays.asList(new Appel("groen", 150),
                               new Appel("rood", 100),
                               new Appel("groen", 200),
                               new Appel("rood", 50)));

List<Appel> resultaat = new ArrayList<>();
for(Appel appel : appeltjes)
{
    if(appel.gewicht > 50)
        resultaat.add(appel);
    }
}
System.out.println(resultaat);
```

Filteren van een verzameling

In veel andere programmeertalen hebben verzamelingen/lijsten een **filter** functie die alle elementen die `false` returnen eruit filtert, en sinds Java 8 dus ook als onderdeel van de **Stream** klasse:

```
List<Appel> zware_appeltjes = appeltjes.stream()
                                       .filter(appel -> appel.gewicht > 50 )
                                       .collect(Collectors.toList());
System.out.println(zware_appeltjes);
```

```
[groen: 150 gram, rood: 100 gram, groen: 200 gram]
```

Hierboven is "`appel -> appel.gewicht > 50`" een anonieme functie die als argument een Appel neemt, en een boolean teruggeeft.

Elementen uit een verzameling halen

Naast filter hebben veel talen ook **map** functie waarmee je bijvoorbeeld alle gewichten in één lijst kan krijgen:

```
List<Integer> gewichten = appeltjes.stream()
                                   .map(appel -> appel.gewicht)
                                   .collect(Collectors.toList());
System.out.println(gewichten);
```

```
[150, 100, 200, 50]
```

Vergelijkbaar voorbeeld in Python

```
class Appel:
    def __init__(self, kleur, gewicht):
        self.kleur = kleur
        self.gewicht = gewicht

    def __repr__(self):
        return "{}: {}gram".format(self.kleur, self.gewicht)

appeltjes = [Appel("groen", 150), Appel("rood", 100),
             Appel("groen", 200), Appel("rood", 50)]

zware_appeltjes = filter(lambda a : a.gewicht > 50, appeltjes)
gewichten = map(lambda a : a.kleur, appeltjes)
```

Inhoud

Er is Java en er is nog meer..

Afgelopen weken hebben we best veel Java-features voorbij zien komen, maar er zijn ook dingen concepten die in andere imperatieve talen wel voorkomen, maar niet in Java.

- Optional/Default arguments
- Named parameters
- Operator overloading
- ... en meer!

Optional/Default arguments

Optional/Default arguments staan het toe om argumenten een standaardwaarde mee te geven. De functie kan dan worden aangeroepen met minder argumenten!

```
int telOp(int a, int b, int c = 0)
{
    return a + b + c;
}
```

Bovenstaande functie kan dus zowel als `telOp(1,2)` als `telOp(1,2,3)` worden aangeroepen.

Named parameters

Argumenten volgen een vaste volgorde, maar dat is in sommige gevallen foutgevoellig:

```
double bmi(double gewicht, double hoogte) {
    return gewicht / (hoogte * hoogte);
}
```

In C# kun je ook deze methode als volgt aanroepen:

```
bmi(80, 1.80);
bmi(hoogte: 1.80, gewicht: 80);
bmi(80, hoogte: 1.80);
```

In Python is iets vergelijkbaars mogelijk, alleen is de syntax wel weer anders.

Operator overloading

Programmeertalen met ondersteuning voor operator overloading staan het toe om operatoren als "+" te definiëren voor zelfgeschreven objecten. Bijvoorbeeld in C#:

```
class Breuk {
    // breuk-code van opgave 3

    public static Breuk operator +(Breuk b1, Breuk b2) {
        return b1.telOp(b2);
    }
}
```

Vervolgens kun je dan gewoon aanroepen:

```
Breuk b3 = new Breuk(1,2) + new Breuk(3,4);
```