Inhoudsopgave: Datum:

[1 Hoofdstuk 1: Zoeken en sorteren 3](#_Toc186124572)

[1.1 Zoeken in een array 3](#_Toc186124573)

[1.2 Linear of sequentieel zoeken 3](#_Toc186124574)

[1.3 Binair zoeken 3](#_Toc186124575)

[1.4 Tijdscomplexiteit 5](#_Toc186124576)

[1.5 Sorteren door selectie 6](#_Toc186124577)

[1.6 Sorteren door tussenvoegen 8](#_Toc186124578)

[1.7 Sorteren door samenvoegen 9](#_Toc186124579)

[2 Hoofdstuk 2: Gelinkte Lijsten 11](#_Toc186124580)

[2.1 Gelinkte lijsten 11](#_Toc186124581)

[2.1.1 Specificatie 11](#_Toc186124582)

[2.1.2 Gelinkte lijst 12](#_Toc186124583)

[2.1.3 Ankercomponenten 14](#_Toc186124584)

[2.1.4 Dubbelgelinkte lijsten 14](#_Toc186124585)

[2.2 Stapels 15](#_Toc186124586)

[2.3 Oefeningen 20](#_Toc186124587)

[3 Hoofdstuk 3: Hashtabellen 23](#_Toc186124588)

[3.1 Inleiding 23](#_Toc186124589)

[3.2 Specificatie 23](#_Toc186124590)

[3.3 Verwerken van de overlappingen 24](#_Toc186124591)

[3.4 Gesloten hashing 24](#_Toc186124592)

[3.5 Open hashing 25](#_Toc186124593)

[3.6 Keuze van hashcode en hashfunctie 26](#_Toc186124594)

[3.7 Oefeningen 28](#_Toc186124595)

[4 Hoofdstuk 4 Bomen: 29](#_Toc186124596)

[4.1 Boom 29](#_Toc186124597)

[4.2 Binaire boom 31](#_Toc186124598)

[4.3 Binaire zoekboom 34](#_Toc186124599)

[4.4 Biniare hoop 36](#_Toc186124600)

[4.5 Binaire hoop 38](#_Toc186124601)

[4.6 Oefeningen: 40](#_Toc186124602)

[5 Hoofdstuk 5: Grafen 41](#_Toc186124603)

[6 Hoofdstuk 6: Zoekalgoritmes 52](#_Toc186124604)

[6.1 Inleiding 52](#_Toc186124605)

[7 Hoofdstuk 7: Complexiteitstheorie 65](#_Toc186124606)

[7.1 Klassa P 65](#_Toc186124607)

[7.2 Klasse NP 66](#_Toc186124608)

[7.3 Tijdscomplexiteit in volgorde 68](#_Toc186124609)

# Hoofdstuk 1: Zoeken en sorteren

## Zoeken in een array

Zoeken in een array:

* Klassieke array met elementen
* Effect van gesorteerde array? Vb. Woordenboek
  + Staat alfabetisch gesorteerd
* Aanname array: **ophalen element in constante tijd** onafhankelijk van positie
  + Het ophalen van een element **arr[i]** duurt even lang (constant)
* Effect sortering: je hoeft niet meer te zoeken in deel van array dat langs de verkeerde kant van het beschouwde element ligt

## Linear of sequentieel zoeken

Linear of sequentieel zoeken (vanaf begin tot einde van de array tellen):

* Dit is **pseudocode**, dit ga je moeten leren schrijven:
  + **Invoer** Een **item zoekitem dat moet gevonden worden, een array** van items genaamd rij met lengte 𝑛.
  + **Uitvoer** de **index van het eerste element in rij dat gelijk is aan zoekitem** wordt teruggegeven of −1 indien zoekitem niet voorkomt in rij.

1: function zoeksequentieel(zoekitem, rij)

2: 𝑖 ← 0 **# overloopt de posities**

3: while 𝑖 < 𝑛 and rij[𝑖] ≠ zoekitem do

4: 𝑖 ← 𝑖 + 1

5: if 𝑖 = 𝑛 then **# niet gevonden**

6: index ← −1

7: else

8: index ← 𝑖 **# gevonden**

9: return index

10: end function

## Binair zoeken

Binair zoeken:

* Dit werkt **enkel in een gesorteerde array**. Zo zoeken we enkel verder in het **relevante deel** van de array.
  + Beschouw een element, bv. Het middelste element
  + Indien het gezochte element **kleiner** is dan het beschouwde: zoek **links** verder
  + Indien het gezochte element **groter** is dan het beschouwde: zoek **rechts** verder
* Merk op dat dit algoritme zowel **iteratief als recursief** kan werken. In geval van recursie:
  + Basisstap: rij met 1 element
  + Recursieve stap: alle andere gevallen: de rij wordt gehalveerd
* Je stopt met zoeken tot maar 1 element overschiet.

Binair zoeken (iteratief = je voert het uit met een lus):

* **Invoer:** **Een item zoekitem dat moet gevonden worden, een** **gesorteerde array** genaamd rij van items van lengte 𝑛.
* **Uitvoer:** De index van het eerste element in rij dat gelijk is aan zoekitem wordt teruggegeven of −1 indien zoekitem niet voorkomt in rij.

Binair zoeken (iteratief) Vervolg:

1: function zoekbinair(zoekitem, rij)

2: 𝑙 ← 0

3: 𝑟 ← 𝑛 − 1

4: while 𝑙 ≠ 𝑟 do # herhalen totdat slechts één element overblijft

5: 𝑚 ← ⌊𝑙+𝑟 2 ⌋

6: if rij[𝑚] < zoekitem then

7: 𝑙 ← 𝑚 + 1 # in de rechterhelft zoeken

8: else

9: 𝑟 ← 𝑚 # in de linkerhelft zoeken

10: if rij[𝑙] = zoekitem then

11: index ← 𝑙

12: else

13: index ← −1

14: return index

15: end function

Binair zoeken (recursief):

* **Invoer** Een item zoekitem dat moet gevonden worden, een **gesorteerde array** genaamd rij van items van lengte 𝑛.
* **Uitvoer** de index van het eerste element in rij dat gelijk is aan zoekitem wordt teruggegeven of **−1** **indien** zoekitem **niet voorkomt** in rij.

1: function zoekbinair(zoekitem, rij)

2: return zoekrecursief(zoekitem, rij, 0, 𝑛 − 1)

3: end function

Binair zoeken (recursief) Vervolg:

* **Invoer** Een item zoekitem dat moet gevonden worden, een gesorteerde array genaamd rij van items van lengte 𝑛, twee natuurlijke getallen 𝑙 en 𝑟 die het gedeelte van de array aangeven waarin gezocht moet worden.
* **Uitvoer** de index van het eerste element in rij dat gelijk is aan zoekitem wordt teruggegeven indien het element voorkomt tussen rij[𝑙], rij[𝑙 + 1], … , rij[𝑟]. De teruggeven index ligt dan tussen 𝑙 en 𝑟. Er wordt −1 teruggegeven indien zoekitem niet voorkomt tussen de elementen rij[𝑙], rij[𝑙 + 1], … , rij[𝑟].

Binair zoeken (recursief) Vervolg:

1: function zoekrecursief(zoekitem, rij, 𝑙, 𝑟)

2: if 𝑙 = 𝑟 then # basisstap, rij van lengte 1

3: if zoekitem = rij[𝑙] then

4: return 𝑙

5: else

6: return −1

7: else # inductiestap

8: 𝑚 ← ⌊𝑙+𝑟 2 ⌋

9: if rij[𝑚] < zoekitem then

10: return zoekrecursief(zoekitem, rij, 𝑚 + 1, 𝑟) # zoek rechts

11: else

12: return zoekrecursief(zoekitem, rij, 𝑙, 𝑚) # zoek links

Efficiënt zoeken (100% een examenvraag):

* Binair zoeken is ingewikkelder dan sequentieel zoeken.
* Focus op twee zaken:
  + 1. De uitvoeringstijd
  + 2. Het geheugengebruik (RAM)

Constante tijd:

Dit wil zeggen dat ongeacht de lengte van de array, de uitvoeringstijd hetzelfde zal blijven.

## Tijdscomplexiteit

Tijdscomplexiteit (hoe lang duurt het om iets te voeren via wiskunde termen):

* In deze cursus: **focus op uitvoeringstijd**. Geen exacte berekening want:
  + **Verschillende computers**
  + Zelfde computer met **andere processen** actief
  + Steeds **krachtigere programmeertalen**: inschatten aantal instructies wordt complex
* Daarom: asymptotische analyse van de uitvoeringstijd. Dit karakteriseert het gedrag van de uitvoeringstijd voor “grote” waarden van de input.

Tijdscomplexiteit Vervolg:

* Voorbeelden van mogelijk gedrag bij asymptotische analyse:
  + Invoer verdubbelt ⟹ uitvoeringstijd verdubbelt
    - **Lineaire** functie: 𝑇 (𝑛) = 𝑛
  + Invoer verdubbelt ⟹ uitvoeringstijd × 4
    - **Kwadratische** functie: 𝑇 (𝑛) = 𝑛2
  + Invoer + 1 ⟹ uitvoeringstijd × 2.
    - **Exponentiële** functie: 𝑇 (𝑛) = 2𝑛 .
  + Invoer verdubbelt ⟹ uitvoeringstijd + constante.
    - **Logaritmische** functie: 𝑇 (𝑛) = log(𝑛)
  + Invoer wijzigt ⟹ uitvoeringstijd verandert niet
    - **Constante** functie: 𝑇 (𝑛) = 𝑐
  + …
* Bij de analyse van de zoekalgoritmen zien we dat de asymptotische **uitvoeringstijd** **bepaald** wordt door het **aantal vergelijkingen dat wordt uitgevoerd**.

Tijdscomplexiteit sequentieel zoeken:

* Bij sequentieel zoeken kunnen we ons afvragen **hoeveel keer** de volgende vergelijking wordt uitgevoerd: **(rij[𝑖] ≠ zoekitem)**
  + Beste geval? ⟹ 1
    - Gesorteerde array
  + **Slechte geval? ⟹ n**
    - Alles behalve een gesorteerde array
    - Zeggen ze niks, wordt er automatische gesproken van een slechtste geval
  + Gemiddeld geval? ⟹ n ÷ 2 dus 𝑇 (𝑛) = 𝑂(𝑛)
* We noemen dit **lineare tijdscomplexiteit** o**f tijdscomplexiteit van orde 𝑛.**

Tijdscomplexiteit binair zoeken:

* Bij binair zoeken tellen we **hoeveel keer** de vergelijking **rij[𝑚] < zoekitem** wordt uitgevoerd. Stel: 𝑛 = 2𝑘 , met 𝑘 een natuurlijk getal
  + 𝑛 = 1, i.e. 𝑘 = 0 ⟹ 0 keer
  + 𝑛 = 2, i.e. 𝑘 = 1 ⟹ 1 keer
  + 𝑛 = 4, i.e. 𝑘 = 2 ⟹ 2 keer
  + 𝑛 = 8, i.e. 𝑘 = 3 ⟹ 3 keer
    - Stel n = 10 dan ligt k ergens tussen 3 en 4 en weer moeten we niet weten.
* Dus: 𝑛 = 2𝑘 ⟺ 𝑘 = log2 (𝑛).
  + Want we zien hier machten in
* Dit noemen we **logaritmische tijdscomplexiteit**: 𝑇 (𝑛) = **𝑂(log2 (𝑛)).**
  + Log functie is trager dan een lineaire functie

Experiment tijdscomplexiteit:

* Voer een experiment uit met hetzelfde algoritme, met n variërend en meet de uitvoeringstijd bvb.
  + Laat 𝑘 de waardes 1 t.e.m. 14 doorlopen.
  + Genereer een willekeurige rij van lengte 𝑛 = 2𝑘 bestaande uit gehele getallen.
  + Kies 30 random getallen uit deze rij.
  + Voer voor elk van deze 30 random getallen de methode 100 000 keer uit en sla de uitvoeringstijd hiervan (apart) op.
  + Maak een grafiek van de gemiddelde uitvoeringstijd voor elke 𝑛 en lees de orde af
  + Pas eventueel de juiste schaal aan op de verticale of horizontale as

Experiment tijdscomplexiteit Vervolg:

* Afbeelding met lijn, Perceel, diagram, schermopname

  Automatisch gegenereerde beschrijvingUitvoeringstijd van lineair en binair zoeken, resp. In de linker- en rechterfiguur. Merk ook de volledige verschillende schaal op de verticale as bij beide plots.
* Merk de verschillen op
* Wat bij kleine 𝑛?
* En bij grote 𝑛?
* Wat met de variantie?

Wat is een exponentiële as:

Ipv 1 🡪 2 🡪 3 🡪 (liniaire) wordt het 1 🡪 2 🡪 4 🡪 8 🡪 16 🡪 (exponentieel)

## Sorteren door selectie

Basisidee:

* Zoek het grootste element en plaats het achteraan.
* Sorteer de rest van de array.

Sorteren door Selectie:

* **Invoer:** De array 𝑎 is gevuld met lengte n (n-elementen).
* **Uitvoer**: De array 𝑎 is gesorteerd.
* Examenvraag kan zijn, de theorie zoals hierboven en daarvan de pseudocode schrijven

1: function SORTERENDOORSELECTIE(𝑎)

2: for 𝑖 = 𝑛 − 1 … 1 by − 1 do # achteraan starten

3: positie ← 𝑖

4: max ← 𝑎[𝑖]

5: for 𝑗 = 𝑖 − 1 … 0 by − 1 do # 𝑗 doorloopt de deelrij

6: if 𝑎[𝑗] > max then

7: positie ← 𝑗

8: max ← 𝑎[𝑗]

9: 𝑎[positie] ← 𝑎[𝑖] # grootste element wisselen met laatste

10: 𝑎[𝑖] ← max

11: end function

Functie uit de les:

1: function zoekendoorselectie(a, n):

11: for j = n -1 tot 1 met stap -1

12: grootste 🡨 a[0]

3: for i = 1 tot n -1

4: if a[i] > grootste:

5: grootste 🡨 a[i]

6: indexgrootste 🡨 i

7: end for

8: hulp 🡨 a[j-1]

9: a[j-1] 🡨 grootste

10: a[grooste] 🡨 hulp

Sorteren door selectie Vervolg:

* Voorbeeld van hoe selectie werkt 🡪
* De uitvoeringstijd wordt bepaald door het aantal keer dat de vergelijking **𝑎[𝑗] > max** op regel 6 uitgevoerd wordt, of het aantal keer dat de teller 𝑗 wijzigt
* Afbeelding met tekst, schermopname, Lettertype, nummer

  Automatisch gegenereerde beschrijvingWe besluiten **𝑇 (𝑛) = 𝑂(𝑛2)** **(kwadtratisch)**

Afbeelding met tekst, Lettertype, wit, typografie

Automatisch gegenereerde beschrijving

Afbeelding met tekst, Lettertype, schermopname, nummer

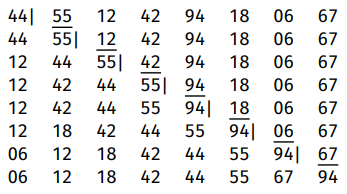
Automatisch gegenereerde beschrijvingHoe Werd die Som Bepaald?

Voorbeeld: bereken: 1 + 2 + 3 + ⋯ + 10.

Dus: 1 + 2 + 3 + ⋯ + 10 =

Algemeen: 1 + 2 + 3 + ⋯ + 𝑚 =

## Sorteren door tussenvoegen

Basisidee:

* Veronderstel dat er reeds een deel vooraan de array gesorteerd is.
* Neem het **eerste element van het niet gesorteerde deel** en **voeg dit element toe op de juiste plaats in het gesorteerde deel**. Op deze manier wordt het gesorteerde deel uitgebreid.
* Sorteren door tussenvoegen of **“card sort“** kan m.a.w. Het best vergeleken worden met het op **volgorde steken van kaarten.**
* Ook genoemd **insertion sort**

Voorbeeld:

* **Invoer** De array 𝑎 is gevuld met 𝑛 elementen.
* **Uitvoer** De array 𝑎 is gesorteerd.

1: function CARDSORT(𝑎)

2: for 𝑖 = 1 … 𝑛 − 1 do

3: 𝑥 ← 𝑎[𝑖] # 𝑥 bevat het in te voegen element

4: 𝑗 ← 𝑖 # 𝑗 zoekt de juiste positie voor 𝑥

5: while 𝑗 > 0 and 𝑥 < 𝑎[𝑗 − 1] do # schuif grotere elementen op

6: 𝑎[𝑗] ← 𝑎[𝑗 − 1] # schuif 𝑎[𝑗 − 1] eentje op

7: 𝑗 ← 𝑗 − 1

8: 𝑎[𝑗] ← 𝑥 # 𝑥 wordt op de juiste positie tussengevoegd

* Mogelijke examenvraag: schrijf cardsort zodat hij van achter naar voor sorteert ipv van voorn naar achter

Complexiteitsanalyse:

* Uitvoeringstijd afhankelijk van ordening bij aanvang
  + Slechtste geval: binnenste lus 𝑛(𝑛−1) 2 keer uitgevoerd.
    - Dus 𝑇 (𝑛) = 𝑂(𝑛\*2 )
  + Beste geval: buitenste lus (𝑛 − 1) keer, maar binnenste lus stopt telkens onmiddellijk.
    - Dus 𝑇 (𝑛) = 𝑂(𝑛)
  + Gemiddeld geval:
    - 𝑇 (𝑛) = 𝑂(𝑛2 ).
* ⟹ **kwadratische tijdscomplexiteit**, want steeds worst case analyse.

## Sorteren door samenvoegen

Sorteren door samenvoegen:

* Sorteren door samenvoegen (mergesort) is een **ingewikkelder algoritme** dan de voorgaande twee eenvoudige sorteeralgoritmes maar is ook een heel stuk efficiënter. Basisidee:
  + 1. **Sorteer** de **eerste helft** van de array.
  + 2. **Sorteer** de **tweede helft** van de array.
  + 3. **Meng** de twee gesorteerde **deelrijen samen** **tot één** gesorteerde array.
* De eerste twee stappen in dit proces **gebeuren** op een **recursieve** manier. Merk op dat het eigenlijk sorteren gebeurt bij het samenvoegen van de twee gesorteerde rijen.

Afbeelding met tekst, schermopname, Lettertype, nummer

Automatisch gegenereerde beschrijvingSorteren door samenvoegen:

* Voorbeeld: Beschouw de rij
  + 44 55 12 42 94 18 06 67
* Deze wordt opgesplitst in **twee deelrijen**
  + 44 55 12 42
  + 94 18 06 67
* Beide deelrijen worden vervolgens **recursief gesorteerd** tot
  + 12 42 44 55
  + 06 18 67 94
* Tot slot worden de gesorteerde **deelrijen** **samengevoegd** tot de gesorteerde rij
  + 06 12 18 42 44 55 67 94

Samenvoegen van twee gesorteerde (deel)rijen:

* De functie MERGESORT roept de recursieve functie MERGESORTRECURSIVE aan. In deze methode wordt een deel van de rij gesorteerd door het te sorteren deel op te splitsen in twee deelrijen van halve lengte. Vervolgens worden de gesorteerde deelrijen gemengd. Het samenvoegen van de beide deelrijen gebeurt in de functie MERGE.
* **Invoer** de array 𝑎 is gevuld met 𝑛 elementen.
* **Uitvoer** de array 𝑎 is gesorteerd

Samenvoegen van twee gesorteerde (deel)rijen Vervolg:

* **Invoer** de array 𝑎 is gevuld met 𝑛 elementen, begin en einde wijzen naar geldige posities in de array 𝑎.
* **Uitvoer** de elementen met index begin tot en met index einde werden gesorteerd.
* **Hier is een geen basisstap, die zit in de else, maar je moet niks doen dus er wordt niks geschreven**

1: function MERGESORTRECURSIVE ((𝑎, begin, einde)

2: if begin < einde then

3: midden ← ⌊(begin + einde)/2)⌋

4: MERGESORTRECURSIVE (𝑎, begin, midden)

5: MERGESORTRECURSIVE (𝑎, midden + 1, einde)

6: MERGE (𝑎, begin, midden, einde)

7: end function

De functie merge:

* **Invoer** de array 𝑎 is gevuld met 𝑛 elementen; de elementen van de deelrij gaande van de **begin-positie** tot en met de **midden-positie** zijn gesorteerd; de elementen van de deelrij gaande van de (**midden+1)-positie** tot en met de **eind-positie** zijn gesorteerd.
* **Uitvoer** de elementen met index begin t.e.m. Index einde werden gesorteerd.

1: function MERGE(𝑎, begin, midden, einde)

2: 𝑖 ← begin # de teller 𝑖 doorloopt de linkse deelrij

3: 𝑗 ← midden + 1 # de teller 𝑗 doorloopt de rechtse deelrij

4: 𝑘 ← 𝑖 # de teller 𝑘 doorloopt de hulparray hulpa

5: hulpa ← nieuwe array[𝑛] # tijdelijke hulpopslag

6: while 𝑖 ≤ midden and 𝑗 ≤ einde do # totdat een deelrij leeg is

7: if 𝑎[𝑖] ≤ 𝑎[𝑗] then # het kleinste element komt eerst

8: hulpa[𝑘] ← 𝑎[𝑖] ; 𝑖 ← 𝑖 + 1

9: else

10: hulpa[𝑘] ← 𝑎[𝑗] ; 𝑗 ← 𝑗 + 1

11: 𝑘 ← 𝑘 + 1 # er is een element in hulpa opgeslagen

12: if 𝑖 > midden then # de 2de deelrij moet leeggemaakt worden

13: while 𝑗 ≤ einde do

14: hulpa[𝑘] ← 𝑎[𝑗] ; 𝑗 ← 𝑗 + 1 ; 𝑘 ← 𝑘 + 1

15: else # de 1ste deelrij moet leeggemaakt worden

16: while 𝑖 ≤ midden do

17: hulpa[𝑘] ← 𝑎[𝑖] ; 𝑖 ← 𝑖 + 1 ; 𝑘 ← 𝑘 + 1

18: for 𝑘 = begin … einde do # kopieer gesorteerde deelrij naar 𝑎 .

19: 𝑎[𝑘] ← hulpa[𝑘] 36/49

20: end function

Complexiteitsanalyse:

* Merge methode: samenvoegen van deelrijen met **lengte 𝑛/2 ⟹ lineair in 𝑛**
* Stel nu 𝑛 = 2𝑘 met 𝑘 een natuurlijk getal
* Mergesortrecursive methode: als 𝑛 = 1 dan is begin = einde ⟹ vgl wordt 1 keer doorlopen
  + 𝑇 (1) = 1,
* Als 𝑛 = 2 dan
  + 𝑇 (2) = 𝑇 (1) + 𝑇 (1) + 2(𝑠𝑎𝑚𝑒𝑛𝑣𝑜𝑒𝑔𝑒𝑛) = 4
  + 𝑇(4) = 2𝑇 (2) + 4 = 2 × 4 + 4 = 12,
  + 𝑇(8) = 2𝑇 (4) + 8 = 2 × 12 + 8 = 32.
* Hieruit kunnen we herkennen dat het algemene patroon is:
  + 𝑇 (𝑛) = 𝑇 (2𝑘 ) = 𝑛 × (𝑘 + 1).
* Dus **tijdscomplexiteit** 𝑇 (𝑛) van sorteren door samenvoegen:
  + **𝑇 (𝑛) = 𝑂(𝑛 log2(𝑛)).**

Geheugengebruik:

* Mergesort is dus tijdsefficiënter dan voorgaande
* Toch minder geheugen efficiënt
* MERGE-algoritme: extra hulprij nodig

# Hoofdstuk 2: Gelinkte Lijsten

## Gelinkte lijsten

### Specificatie

Inleiding:

* Array = eenvoudige datastructuur
* Maar wat met elementen tussenvoegen? Of verwijderen in het midden?
* **Bewerkingen hebben lineaire tijdscomplexiteit**
* Oplossing? ⟹ **Gelinkte lijsten**
* **Toevoegen of verwijderen in constante tijd (dankzij gelinkte lijst)**
* **Opzoeken** van een element **blijft** echter een **lineare** tijdscomplexiteit vertonen
* Nog een voordeel: aantal elementen toevoegen kan onbeperkt, geen limieten op de grootte zoals bij array

Definitie:

* Een gelinkte lijst bestaat uit een aantal **knopen** die via een **kettingstructuur** aan elkaar geschakeld zijn. Een knoop bestaat uit twee velden:
  + Een data-veld **data**
  + Een veld **volgende**
* De laatste knoop bevat een wijzer **null**: dit wordt grafisch voorgesteld door een schuine streep. Voor de eerste knoop moet een referentie eerste bijgehouden worden. In een lege lijst is de referentie **eerste** gelijk aan null.

Een eenvoudig geschakelde lijst:

Afbeelding met diagram, lijn, Lettertype, Technische tekening

Automatisch gegenereerde beschrijving

Basisbewerkingen:

* De belangrijkste basisbewerkingen voor een enkelvoudig geschakelde lijst zijn:
  + **Zoek():** zoekt de positie van de knoop met als data-veld het argument;
  + **Verwijder():** verwijdert de knoop die volgt na de opgegeven knoop en geeft de waarde van het data-veld van de verwijderde knoop weer;
  + **Voegtoe():** voegt een knoop toe na een opgegeven knoop, het data-veld krijgt de waarde van het tweede argument.

Afbeelding met tekst, Lettertype, schermopname, lijn

Automatisch gegenereerde beschrijvingDe klasse Knoop in UML:

De gedefinieerde Knoop is een datastructuur die een element van een niet nader gedefinieerde klasse Element bevat.

Constructor voor een Knoop:

* **Invoer** /
* **Uitvoer** er werd een nieuwe knoop aangemaakt

1: function knoop

2: data ← null

3: volgende ← null

### Gelinkte lijst

Implementatie Gelinkte Lijst:

De klasse Knoop wordt als **inwendige klasse** (inner class) van de klasse gelinktelijst geïmplementeerd. Dit betekent dat methodes van de klasse gelinktelijst toegang hebben tot de velden van Knoop. We schrijven de klasse gelinktelijst uit in UML-notatie. Alle methodes worden in pseudocode beschreven.

De klasse gelinktelijst in UML:

Afbeelding met tekst, Lettertype, schermopname, lijn

Automatisch gegenereerde beschrijving

Algoritme voor de Constructor:

* De constructor gelinktelijst maakt een nieuw object van de klasse gelinktelijst aan.
* **Invoer** /
* **Uitvoer** er werd een nieuwe (lege) gelinkte lijst aangemaakt

1: function gelinktelijst

2: eerste ← null

3: end function

Opzoeken van een element 𝑥:

* **Invoer** de gelinkte lijst werd aangemaakt, 𝑥 is het te zoeken element
* **Uitvoer** de referentie naar de eerste knoop met dataveld gelijk aan 𝑥 werd geretourneerd, indien 𝑥 niet voorkomt in de lijst werd de referentie null geretourneerd.

1: function ZOEK(𝑥)

2: ref ← eerste

3: while ref ≠ null and ref.data ≠ 𝑥 do

4: ref ← ref.volgende

5: return ref

6: end function

Tijdscomplexiteit van ”zoek”:

* De uitvoeringstijd voor de methode zoek is afhankelijk van de positie van 𝑥 in de lijst.
  + Slechtste geval: elke knoop van de lijst overlopen en 𝑥 niet gevonden ⟹ lineaire uitvoeringstijd.
  + Beste geval: gezochte referentie onmiddellijk gevonden ⟹ constante uitvoeringstijd.
  + Gemiddeld geval: de helft van de knopen overlopen ⟹ lineaire uitvoeringstijd.

Vervolg tijdscomplexiteit:

* Zoeken van een element 𝑥 in een gelinkte lijst is niet efficiënter dan in een gewone array.
* Integendeel, in de praktijk is het volgende van referenties (pointers) trager dan het doorlopen van een array.
* De overige basisfuncties voor een gelinkte lijst zullen wel allemaal uitvoerbaar zijn in een constante tijd, op voorwaarde dat de referentie reeds gekend is.

Afbeelding met diagram, lijn, Plan, Technische tekening

Automatisch gegenereerde beschrijvingVerwijderen van een knoop:

* De functie VERWIJDER verwacht als **inputwaarde** een **referentie ref**. Deze referentie ref verwijst naar de knoop in de gelinkte lijst die de te verwijderen knoop voorafgaat en de waarde van het data-veld van de verwijderde knoop wordt geretourneerd. Waarom argument ref nodig? En waarom ref naar voorgaande knoop?
* **Invoer:** de gelinkte lijst 𝑙 bestaat, de knoop ref is niet de laatste knoop in de lijst.
* **Uitvoer:** de knoop die volgt na de knoop met referentie ref werd verwijderd uit de lijst, het data-veld van de verwijderde knoop werd geretourneerd.

1: function VERWIJDER(ref)

2: 𝑥 ← ref.volgende.data

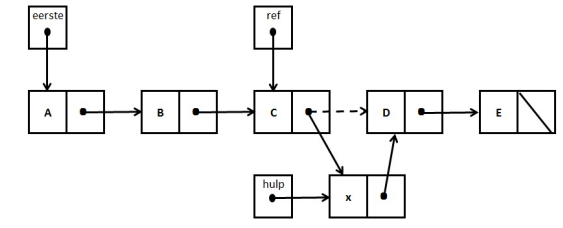
3: ref.volgende ← ref.volgende.volgende

4: return 𝑥

5: end function

Vooraan verwijderen:

Wat als we nu de eerste knoop willen verwijderen? ⟹ **is niet mogelijk**, want je kan de voorgaande van de eerste niet meegeven als argument. Dit probleem wordt later aangepakt (zie ankercomponenten).

Toevoegen van een knoop:

* De methode VOEGTOE voegt een nieuwe knoop met data-veld 𝑥 toe na de knoop ref.
* Creëer een nieuwe knoop **hulp**;
* Stockeer de waarde **𝑥** in het data-veld van de knoop **hulp**;
* Laat **hulp.volgende** verwijzen naar **ref.volgende**;
* Laat **ref.volgende** verwijzen naar **hulp**.
* **Invoer** de gelinkte lijst bestaat, en ref is niet null.
* **Uitvoer** na de knoop, waarnaar gerefereerd wordt door de referentie ref, werd een nieuwe knoop met data-veld 𝑥 toegevoegd.

1: function voegtoe(ref, 𝑥)

2: hulp ← nieuwe Knoop( )

3: hulp.data ← 𝑥

4: hulp.volgende ← ref.volgende

5: ref.volgende ← hulp

6: end function

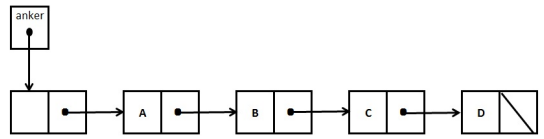
Vooraan toevoegen:

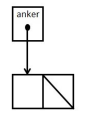
* Wat als we een knoop helemaal vooraan willen toevoegen? ⟹ **niet mogelijk**, want je kan de voorgaande knoop niet meegeven als argument.
* Wat met een lege lijst? ⟹ niet mogelijk
* Ook hier kan een **ankercomponent** hulp bieden

### Ankercomponenten

Ankercomponenten:

* De methoden VERWIJDER en voegtoe uit de voorgaande paragraaf zijn enkel bruikbaar voor algemene gevallen.
  + **Voegtoe aan lege lijst? ⟹ implementatie niet bruikbaar**
  + **Voegtoe vooraan in lijst?**
  + **Verwijder eerste knoop?**
* Daarom ankercomponenten ingevoerd!
* De ankercomponent is een extra knoop die **helemaal vooraan** aan de gelinkte lijst wordt toegevoegd. Het **data-veld** van deze ankercomponent is **leeg**. Het **referentie**-veld van de ankercomponent verwijst naar de **eerste knoop** van de gelinkte lijst. De ankercomponent maakt dus logisch gezien geen deel uit van de eigenlijke lijst maar dient enkel om de implementatie te vereenvoudigen.

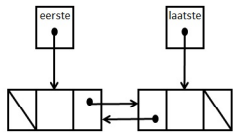
Gelinkte lijst met ankercomponent:

* Nu heeft elke knoop dus een **voorganger** in de lijst. Problemen opgelost! Een lege gelinkte lijst bestaat nu uit één enkele knoop, nl. De ankercomponent, en de referentie naar deze component. Hieronder een lege gelinkte lijst met ankercomponent.
* Het gebruik van een ankercomponent maakt controle op speciale situaties overbodig ⟹ vereenvoudigt de basisbewerkingen

### Dubbelgelinkte lijsten

Dubbelgelinkte lijsten:

* **Eerste knoop vlot bereikbaar**
* Wat als je de laatste wil bereiken ⟹ Tijd!
* Oplossing? Een ref vorige en volgende …
* Dan ook direct referentie laatste knoop bijhouden? Ja!
* Dit heet een **dubbelgelinkte lijst**

Ankercomponenten:

* Een lege dubbelgelinkte lijst met **twee ankercomponenten**.
* **Testen of de lijst leeg is**, kan als volgt:
  + Als **eerste.volgende = laatste**
    - Of
  + Als **laatste.vorige = eerste**.

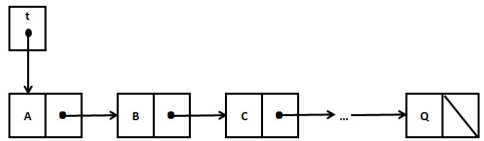
## Stapels

Stapel:

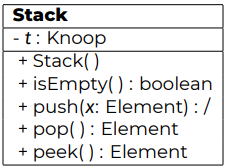
* De naam stapel of stack is gekozen naar analogie met een stapel boeken.
* Indien we een boek van de stapel nodig hebben dan is het best bereikbare boek hetgeen bovenaan ligt. Een boek dat zich op een andere plaats bevindt, is slechts bereikbaar als we alle bovenliggende boeken verwijderd hebben.
* Ditzelfde principe wordt toegepast voor het datatype stack:
  + Een element dat we willen toevoegen aan een stapel, komt steeds bovenop de reeds bestaande stapel te liggen.
  + Enkel het bovenste element van de stapel kan verwijderd worden. Dit element wordt de top van de stapel genoemd.
* Een stapel is m.a.w. Een LIFO- of Last-In-First-Out-structuur.

Specificatie:

* De voorgaande beschrijving legt de structuur van een stapel vast. Voor de implementatie van deze structuur zullen we gebruik maken van een klasse Stack. In deze klasse worden een aantal basisbewerkingen gedefinieerd. Deze zijn:
  + **Stack():** **constructor**, maakt een nieuwe stapel aan waarna de stapel bestaat als lege stapel;
  + **Isempty():** controleert of een stapel al dan niet **leeg** is;
  + **Push():** voegt een **nieuw element** toe bovenaan een stapel, het toegevoegde element wordt de **nieuwe top** van de stapel;
  + **Pop():** **verwijdert** het **bovenste element** van een stapel en **retourneert** het verwijderde element;
  + **Peek():** geeft het **bovenste element** van de stapel terug, zonder het te verwijderen.
* Vertaling van deze basisbewerkingen nodig naar ondubbelzinnige algoritmen met efficiënte implementatie ⟹ **uitvoeringstijd niet afhankelijk van de grootte van de stapel.**

Implementatie van een Stapel:

We tonen nu hoe een stapel kan geïmplementeerd worden als een gelinkte lijst waarbij enkel de top bereikbaar is. Het gebruik van een ankercomponent is hier niet nodig. In het bijzonder wordt een stapel geïmplementeerd door een referentie **𝑡 naar de eerste knoop in de lijst**, die de top van de stapel bevat. De top van de stapel bevat het element 𝐴. Het element 𝑄 werd als eerste element op de stapel geplaatst.

De klasse Stack in UML:

De klasse Knoop wordt als inwendige klasse (inner class) van de klasse Stack geïmplementeerd. Dit betekent dat methodes van de klasse Stack toegang hebben tot de velden van Knoop. De implementatie van de klasse Knoop is zoals voorheen.

De Stack Constructor:

* Een lege stapel bevat nog geen elementen, m.a.w. De referentie 𝑡 is null.
* **Invoer** /
* **Uitvoer** er werd een nieuwe stapel aangemaakt, deze stapel bestaat als lege stapel.

1: function STACK

2: 𝑡 ← null

3: end function

De methode isempty():

* De methode ISEMPTY controleert of een stapel al dan niet leeg is. Het resultaat van de methode is een boolean.
* **Invoer** de stapel 𝑠 bestaat
* **Uitvoer** de waarde **true** of **false** werd afgeleverd, afhankelijk van het feit of de stapel 𝑠 leeg is of niet.

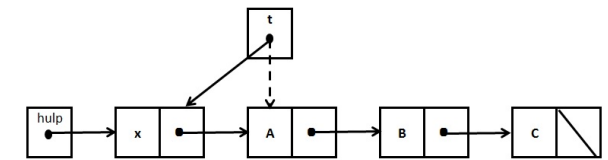
1: function isempty

2: return 𝑡 = null # vergelijking, geen assignatie of return non t

3: end function

Toevoegen van een element:

* **Invoer** de stapel 𝑠 bestaat, 𝑥 moet op de stapel worden geplaatst
* **Uitvoer** het element 𝑥 werd als top-element op de stapel 𝑠 geplaatst.

1: function push(𝑥)

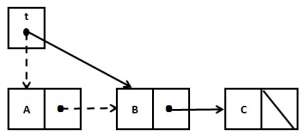
2: hulp ← nieuwe Knoop( )

3: hulp.data ← 𝑥

4: hulp.volgende ← 𝑡

5: 𝑡 ← hulp

6: end function

Verwijderen van een element:

* **Invoer** de stapel 𝑠 bestaat en is niet leeg
* **Uitvoer** de top werd verwijderd en de waarde van de top werd geretourneerd.

1: function POP

2: 𝑥 ← t.data

3: 𝑡 ← t.volgende

4: return 𝑥

5: end function

Retourneren van de waarde van de top van een stapel:

* Er wordt opnieuw verondersteld dat de stapel niet leeg is.
* **Invoer** de stapel 𝑠 bestaat en is niet leeg U
* **Uitvoer** de waarde van de top werd geretourneerd, 𝑠 werd niet gewijzigd.

1: function PEEK

2: return t.data

3: end function

Controleren van haakjes:

* Elke programmeur kent het verschijnsel dat een programma niet werkt omdat ergens een haakje ontbreekt. Compilers controleren steeds of de haakjes allemaal correct zijn en genereren een foutboodschap wanneer dit niet het geval is.
* We tonen nu aan hoe een stapel kan gebruikt worden om te controleren of de haakjes correct zijn. Het basisidee is dat men voor deze controle enkel het laatste openende haakje nodig heeft.
* **Basisidee**: Voor de controle van de haakjes worden twee zaken onderzocht:
  + 1. Er wordt gecontroleerd of elk openend haakje juist één overeenkomstig sluitend haakje heeft, bv. Voor elk ‘(’-symbool moeten we verder in het programma één ‘)’-symbool terugvinden. Dit geldt ook voor: [ en ], { en }.
  + 2. De volgorderegels moeten in acht genomen worden. De haakjes mogen genest zijn, dit wil zeggen dat een paar haakjes zich tussen een ander paar mag bevinden, maar paren haakjes mogen elkaar niet overlappen. Bijvoorbeeld, ([ ]) is een geldige volgorde van symbolen, terwijl ([ )] niet geldig is.
* We willen met behulp van een stapel een algoritme ontwerpen dat de vereffening van symbolen verifieert.
* Dit kan als volgt (uitleg pseudocode haakjes):
  + Maak een **lege stapel** aan.
  + **Doorloop alle symbolen** één voor één. Indien het symbool een haakje is, moet er een opdracht uitgevoerd worden:
    - Als het een **open-symbool** is, plaats het dan op de **stapel** (push-bewerking);
    - Als het een **sluit-symbool** is onderscheiden we **twee** mogelijke **situaties**:
      * De **stapel** is **leeg**: er wordt een **foutmelding** gegenereerd aangezien er geen corresponderend open-symbool aan is vooraf gegaan;
      * De **stapel** is **niet leeg**: het top-element wordt er afgehaald (pop-bewerking). Dit symbool wordt vergeleken met het net ingelezen symbool, als beide symbolen niet corresponderen wordt een fout gemeld.
  + Alle karakters ingelezen en stapel niet leeg? Foutmelding: er zijn nog open-symbolen zijn waarvoor geen sluit-symbool is gevonden

Controleren van haakjes: algoritme:

* De pseudocode om te controleren of het om een openend haakje gaat en om na te gaan of haakjes van dezelfde soort zijn wordt niet getoond omdat deze niet tot de essentie van het algoritme behoren
* **Invoer:** uitdrukking is een array van Strings met de tokens van een uitdrukking waarin eventueel haakjes voorkomen.
* **Uitvoer:** indien alle open haakjes correct worden afgesloten werd er geen foutmelding gegenereerd. In het andere geval wordt er een boodschap getoond op het scherm

**Pseudocode controle haakjes:**

1: function CONTROLEERHAAKJES(uitdrukking)

2: 𝑠 ← STACK()

3: for 𝑖 = 0 … uitdrukking.lengte − 1 do

4: symbool ← uitdrukking[𝑖]

5: if symbool is openend haakje then

6: 𝑠.push(symbool)

7: else

8: if symbool is sluitend haakje then

9: if 𝑠.isempty() then

10: print(”Te veel sluit symbolen”)

11: else

12: voorgaand ← 𝑠.POP()

13: if symbool en voorgaand niet corresponderend then

14: PRINT( ”Fout symbool:”, symbool)

15: if not 𝑠.isempty() then

16: print(”Te veel open symbolen.” )

17: end function

Infix en postfix:

* In de wiskunde worden in een rekenkundige uitdrukking de binaire operatoren zoals + en × normaalgezien tussen de operanden geschreven: **1 + 2 × 3**.
* Hierbij staan de operatoren + en × tussen de operanden 1, 2 en 3. Dit heet **infix**-notatie.
* Vertaling door compiler naar machine-code niet rechtstreeks mogelijk
* Oplossing: postfix-notatie
* Andere naam: RPN (Reverse Polish Notation) + (voluit niet kennen).
* Vb. Sommige HP rekenmachines operatoren na hun operanden pas: **3 4 × .**
* De binaire vermenigvuldigingsoperator staat na zijn operanden en werkt in op de twee argumenten ervoor nl. 3 en 4. De betekenis van deze postfix-uitdrukking is dus 3 × 4 en de waarde ervan is 12.
* Nog een voorbeeld: **1 2 3** × +
* Komt overeen met **1 6 +** wat evalueert naar 7.

Infix: Prioriteiten:

Verder moet bij het verwerken van rekenkundige uitdrukkingen eveneens rekening gehouden worden met de prioriteitsregels. Zo moet de vermenigvuldiging uitgevoerd worden vóór de optelling, tenzij haakjes anders aangeven. Ook dit moet correct verwerkt worden door de compiler. De waarde van 1 + 2 × 3 is m.a.w. 7 en niet 9. Om de waarde 9 te bekomen schrijft men (1 + 2) × 3. Hierbij werden haakjes gebruikt die aangeven dat de optelling eerst moet worden uitgevoerd.

Waardebepaling van een rekenkundige uitdrukking:

* Standaard wordt er gewerkt volgens de volgende afspraken:
  + De operatoren **× en /** hebben een **hogere prioriteit** dan **+ en −**
  + **Prioriteiten** kunnen door gebruik van haakjes (in de meeste programmeertalen zijn alleen ronde haakjes hiervoor toegelaten) worden **aangepast**;
  + De prioriteit van **× en / is gelijk**. Als ze beide in een uitdrukking staan, worden ze uitgevoerd in de volgorde waarin je ze tegenkomt, dus van **links naar rechts.** Hetzelfde geldt voor + en −.

Afbeelding met tekst, Lettertype, schermopname, nummer

Automatisch gegenereerde beschrijvingHaakjes:

Tabel: Infix-notatie versus postfix-notatie. Haakjes zijn overbodig in de postfix-notatie.

Prioriteitsregels:

Voor de evaluatie van een postfix-uitdrukking hoeven eveneens geen prioriteitsregels meer in acht genomen te worden. De uitdrukking wordt steeds eenvoudigweg van links naar rechts doorlopen. Wanneer een binaire operator ontmoet wordt, wordt deze uitgevoerd op de beide voorgaande operanden. In de postfix-uitdrukking wordt vervolgens de operator samen met zijn operanden vervangen door het resultaat van de bewerking.

Postfix: waardebepaling:

* Een machine kan de waarde van een postfix-uitdrukking gemakkelijk bepalen met behulp van een stapel:
  + De uitdrukking wordt van links naar rechts doorlopen.
  + Als een operand (i.e. Een getal) wordt ontmoet dan wordt dit op de stapel geplaatst.
  + Als een (binaire) operator wordt ontmoet dan worden de twee laatst gestapelde operanden van de stapel gehaald en de bewerking met die twee operanden wordt uitgevoerd. Het resultaat van de bewerking wordt op de stapel geplaatst.
  + Deze stappen worden herhaald totdat de volledige uitdrukking is doorlopen.
  + Als de volledige uitdrukking is doorlopen bestaat de stapel nog slechts uit één element, nl. De waarde van de postfix-uitdrukking.

Uitvoeringstijd:

* Dit algoritme is uitvoerbaar in **lineaire tijd**, aangezien voor elk symbool uit de uitdrukking maximaal een constant aantal stapel-bewerkingen moet uitgevoerd worden, waarbij elke stapel-bewerking constante tijd vraagt. Dit wordt uitgewerkt in de oefeningen
* Toch worden in de meeste programmeertalen rekenkundige uitdrukkingen in infix-notatie geschreven omdat dit voor de programmeur handiger is. Gelukkig bestaat er een eenvoudig algoritme om een infix-uitdrukking om te zetten naar een equivalente postfix-uitdrukking.

Van infix naar postfix:

* De conversie van een (gewone) infix-uitdrukking als 3 + 4 × 5 naar zijn overeenkomstige postfix-uitdrukking 3 4 5 × + kan eveneens **met behulp van een stapel** gebeuren.
* Uit de voorbeelden van Tabel 1 leren we dat in een postfix-uitdrukking de operanden steeds in precies dezelfde volgorde staan als in de equivalente infix-uitdrukking. De volgorde van de operatoren kan echter wijzigen.

Van infix naar postfix vervolg:

* 1. Lees de invoertekst van links naar rechts.
* 2. Operand? ⟹ schrijf deze rechtstreeks naar de uitvoer
* 3. Operator of haakje ⟹ tijdelijk op de stapel als:
  + De stapel leeg is;
  + De gelezen operator hogere prioriteit dan de operator bovenaan de stapel;
  + Openend haakje (fungeert verder als operator met de laagste prioriteit)
* 4. Operator heeft gelijke of lagere prioriteit dan die op de top van de stapel
  + Haal alle operatoren van de stapel met gelijke of hogere prioriteit en voeg toe aan de uitvoer;
  + Hierna is de stapel leeg of is de top een operator met lagere prioriteit;
  + Vervolgens wordt de ingelezen operator op de stapel geplaatst;
* 5. Sluitend haakje wordt ingelezen
  + Haal alle operatoren van de stapel en voeg toe aan de uitvoer totdat een openend haakje wordt bereikt;
  + Het haakje wordt eveneens van de stapel gehaald maar niet aan de uitvoer toegevoegd;
* 6. Als het einde van de invoertekst bereikt is, worden alle operatoren van de stapel gehaald en aan de uitvoer toegevoegd totdat de stapel leeg is.

## Oefeningen

Oefening 2.2:

* **Invoer**: een gelinkte lijst
* **Uitvoer**: een nieuwe gelinkte lijst met omgekeerde Elementen van de input

Function invert():

Ref <- eerste.volgende()

L2 <- nieuwe gelinktelijst()

While ref != null:

L2.voegtoe(l2.anker, ref.data) # voegt toe na anker

Ref <- ref.volgende() # verwijst naar de volgende knoop zijn data (ref.data) in l1

Examenvraag:

Toevoegen van een knoop met voegtoe(x, y), wanneer er wordt gevraagd ‘er wordt een ankercomponent toegevoegd, verandert deze methode?’ het antwoord kan nee zijn buiten de Knoop constructor

Oefening 3:

**Function dubbelgelinktelijst():**

Eerst <- nieuwe knoopdubbel()

Laastste <- nieuwe knoopdubbel()

Eerste.volgende <- laatste

Laatste.volgende <- eerste

Eerste.vorige <- null

Eerste.

**Function verwijder(ref):**

Ref.vorige.volgende <- ref.volgende

Ref.volgende.vorige <- ref.vorige

Return ref.data

**Function voegtoena(ref, x):**

Hulp <- nieuweknoopdubbel()

Hulp.data <- x

Hulp.vorige <- ref

Hulp.volgende <- ref.volgende

Ref.volgende <- hulp

Hulp.volgende.vorige <- hulp

**Function voegtoevoor(ref, x):**

Voegtoena(ref.vorige, x)

Function zoek(x):

Links <- eerst.volgende

Rechts <-laatste.vorige

While( links != rechts) and (links.data = x or rechts.data = x) and (links.volgende != rechts):

Links <- links.volgende

Rechts <- rechts.vorige

Oefening 4:

**Function queue():**

K <- null # kop

S <- null # staart

**Function isempty():**

Return (k = null)

**Function enqueue(x):**

Hulp <- nieuwe Knoop(x)

Hulp.data <- x

S.volgende <- hulp

If (k = null):

K <- hulp

S <- hulp

**Function dequeue():**

X <- k.data

If isempty()

Return null

If k = s:

K <- null

S <- null

Else:

K <- k.volgende

Return x

**Hulptheorie dequeue/stack:**

* S en K verwijzen naar een lege lijst
* S en K verwijzen beide naar hetzelfde element van een lijst met één element
* S verwijst naar het laatste element en K verwijst naar het eerste element in een gelinkte lijst met 2 of meer elementen.

**Function front():**

If k = null:

Return k.data

Else:

Return null

**Function invert():**

Vorige <- k

Huidige <- k.volgende

Volgende <- k.volgende.volgende

While huidige = null:

Huidige.volgende <- vorige

Vorige <- huidige

Huidige <- volgende

Volgende <- volgende.volgende

Hudige.volgend <- vorige

K.volgende <- null

K, s <- s, k

# Hoofdstuk 3: Hashtabellen

## Inleiding

Inleiding:

* We hebben al kennis gemaakt met een array en gelinkte lijst als gegevensstructuur.
* Element opzoeken in een **gelinkte lijst: lineaire tijd.**
* Andere basisfuncties in een constante tijd: op voorwaarde dat men reeds de positie heeft bepaald en de referentie hiernaar kent
* Stel: we wensen een digitaal woordenboek te implementeren. Dan moet het opzoeken efficiënt gebeuren.
  + Voor dergelijke toepassingen bieden hashtabellen een oplossing.
  + **Voordeel**: typisch hebben we in een hashtabel een constante tijd voor: het opzoeken, toevoegen en het verwijderen van elementen ○
  + **Nadeel**: een hashtabel is echter niet in staat om de elementen gesorteerd terug te geven, wat bv. Bij een binaire zoekboom (zie later) wel het geval is.
  + **En Je verspilt heel veel plaats (bv**. 5, 91, 399, 528 en 739). Hier voor heb je een array nodig tot 738 maar dit is veel geheugenverspilling.

Stelling:

* Stel: een verzameling waarbij de sleutelwaarden de natuurlijke getallen tussen 0 en 999 zijn. In dit geval kan men eenvoudig weg een array 𝑎 van grootte 1000 alloceren, en we spreken af dat **𝑎[𝑖] = 𝑖** enkel en alleen als het getal 𝑖 tot de verzameling hoort.
* We gebruiken een bijzonder waarde (bv. “null” of −1) om aan te geven dat 𝑖 niet tot de verzameling behoort ⟹ de drie basisbewerkingen, nl. Toevoegen, opzoeken en verwijderen kunnen in constante tijd uitgevoerd worden.
* Het gebruik van een array op deze manier noemen we **directe adressering**.
  + We zoeken element 800 op (die op plaats 799 zit) we weten dus direct waar het element zich bevind, dus de zoektijd is constant.

Afbeelding met tekst, Lettertype, nummer, lijn

Automatisch gegenereerde beschrijvingVoorbeeld:

Voorbeeld: de verzameling natuurlijke getallen tussen 0 en 9. Dan komt de array overeen met de verzameling {1, 3, 7, 8, 9}

## Specificatie

Specificatie:

* In veel gevallen zijn de objecten die we willen opslaan echter geen natuurlijke getallen.
  + Stel strings van 2 karakters lang bvb. “aa” t.e.m. “zz”.
  + Construeer afbeelding die tweeletterige string afbeeldt op uniek natuurlijk getal 0 − 675. Vb 𝑓(𝑎) = 0, 𝑓(𝑧) = 25
  + Daarna interpreteren we deze twee cijfers in basis 26. De index voor het woord “je” is dan gelijk **aan 9 × 261 + 4 × 260 = 238** (10e letter en 5e letter = je).
* Deelverzamelingen van alle tweeletterige strings kunnen m.a.w. Eenvoudig geïmplementeerd worden m.b.v. Een array van lengte 26 × 26 = 676. Opnieuw moet afgesproken worden welke speciale waarde wordt gebruikt om aan te geven dat een element niet tot de verzameling behoort.

Zelfde tactiek voor het woord “hashtabel” :

* 7 × 268 **(h)** + 0 × 267 **(a)**+ 18 × 266 + 7 × 265 + 19 × 264 + 0 × 263 + 1 × 262 + 4 × 261 + 11 × 260 = 1467 441 788 967 ≈ 1.46 × 1012 .
* Veel te grote getallen!
* Meeste elementen toch “null” want meeste combinaties van letters zijn geen geldige woorden
* Oplossing: Kies grootte 𝑁 en bereken **HASHCODE**
* Transformeer hashcodes naar bereik tussen 0 en 𝑁 − 1 door gebruik HASHFUNCTIE. Vb. Modulo 𝑁, i.e. Men neemt de rest na deling door 𝑁.

Functie:

* Formeel uitgedrukt, waarbij 𝑤 een woord voorstelt, vinden we
* ℎ(𝑤) = 𝑤.hashcode( ) (mod 𝑁).
* Deze hashfunctie zorgt er inderdaad voor dat we voor elk woord (of object) 𝑤 een positie bepalen in de array.

## Verwerken van de overlappingen

Botsingen (zelfde positie voor andere elementen – waar of niet waar examenvraag):

* Aangezien er meer hashcodes zijn dan posities in de tabel is het niet uitgesloten dat een aantal woorden op dezelfde positie moet opgeslagen worden. De opbouw van de hashtabel zal afhangen van de manier waarop wordt omgegaan met deze **botsingen**. Algemeen kunnen we stellen dat hashing kan opgesplitst worden in twee luiken:
  + 1. De keuze van een hashfunctie ℎ die alle mogelijke items afbeeldt op een positie uit de hashtabel.
  + 2. Het selecteren van een methode om de waarden die overlappen of botsen (collisions) te verwerken.

Types hashing:

* Er zijn verschillende oplossingsmethodes voor het opvangen van de collisions. We bespreken twee methodes.
  + 1. **Gesloten hashing**: zoek in de tabel zelf naar een goede positie om een element, dat botst met een ander element uit de tabel, op te slaan.
  + 2. **Open hashing**: zoek buiten de tabel naar een plaats om de overlappende elementen op te slaan (steek het op de volgende positie)

## Gesloten hashing

Gesloten hashing:

* In het geval van GESLOTEN HASHING worden alle woorden in de array zelf opgeslagen.
* Er doet zich een BOTSING voor wanneer de positie die correspondeert met een bepaald woord reeds ingenomen is, of anders gezegd ℎ(𝑤) is reeds bezet.
* Het woord 𝑤 kan dan niet op die positie opgeslagen worden zonder informatie te verliezen
* Alternatieve positie om het woord op te slaan kan gekozen worden voor de eerstvolgende vrije positie in de tabel, waarbij de tabel als een circulaire structuur wordt opgevat
  + ⟹ LINEAIRE PEILING.

Lineaire peiling: voorbeeld:

* We bouwen een hashtabel op van lengte 𝑁 = 10. In de tabel moeten een aantal gehele getallen opgeslagen worden. Als bijhorende hashcode kiezen we voor het getal zelf. Als hashfunctie kiezen we voor
* ℎ(𝑤) = 𝑤 (mod 𝑁).
* We voegen één voor één de elementen 10, 15, 29, 100, 115 en 129 toe. Indien een positie reeds ingenomen is, zoeken we naar de eerste vrije positie om het nieuwe element op te slaan. Hierbij beschouwen we de tabel als een circulaire structuur: de eerste positie volgt op de laatste positie. (129 % 10 = 9. T9 is bezet, t0 ook, t1 ook, t2 niet dus daar komt 129)

Afbeelding met tekst, schermopname, Lettertype, nummer

Automatisch gegenereerde beschrijving

Ook zoeken gaat eenvoudig:

* Bereken hashfunctie voor element
* Indien leeg: element niet aanwezig
* Indien bezet: ofwel element gevonden
* Indien bezet: element zoeken op volgende posities
  + Tot gevonden
  + Of lege positie
  + Of terug bij beginpositie

Efficiëntie van zoeken is afhankelijk van het aantal lege posities in de tabel (lineaire peiling):

* Bij toename aantal elementen
  + Groeperen : primaire clustering genoemd
* Wat als we elementen zomaar verwijderen?
  + Kan gat creëren in sequentie vb. Verwijder waarde 10
* Daarom plaatsen we een vlag om te duiden dat het element verwijderd is.

## Open hashing

Open hashing:

* In het geval van open hashing wordt er buiten de tabel gezocht naar een goede positie om elementen op te slaan.
  + Mogelijke oplossing : sla alle elementen op dezelfde positie in de hashtabel op in een gelinkte lijst.
  + Afbeelding met diagram, schets, Technische tekening, Plan

    Automatisch gegenereerde beschrijvingIn praktijk bestaat zo’n hashtabel uit een array van lengte 𝑁 en op elke positie van de array wordt een referentie opgeslagen naar de gelinkte lijst met alle waarden
  + Je nieuwste woord moet altijd **vooraan** staan wegens de efficiëntie

Voorbeeld:

Voorbeeld van de map interface voor de definitie van woorden. Om te kunnen achterhalen welke knoop van de gelinkte lijst bij welk woord hoort, zal in de knopen van de gelinkte lijst naast de definitie ook het woord zelf moeten bijgehouden worden.

Implementatie van open hashing:

* Veronderstel dat we een woordenboek willen opbouwen waarbij we met een groot aantal woorden 𝑤 een definitie willen associëren. In het geval van open hashing zijn basisfuncties voor een hashtabel als volgt:
  + **Hashfunctie(𝑤)** voert achtereenvolgens uit:
    - Bereken de hashcode van het woord 𝑤;
    - Bereken de bijhorende positie;
    - Geef de berekende positie terug.
  + **Voegtoe(𝑤,definitie)** voert achtereenvolgens uit:
    - Bereken de juiste positie voor het woord 𝑤 met de hashfunctie;
    - Maak een nieuwe knoop aan met drie velden:
      * Een veld voor het woord,
      * Een veld voor de definitie,
      * Een veld volgende;
    - Voeg de knoop toe aan de gelinkte lijst die start op de gevonden positie.
  + **Zoekop(𝑤)** werkt als volgt:
    - Bereken de juiste positie van het woord 𝑤 met de hashfunctie;
    - Zoek in de gelinkte lijst, waar de gevonden positie naar refereert, naar de knoop met het woord 𝑤;
    - Geef de definitie van die knoop terug of geef aan dat het gegeven woord niet tot de hashtabel behoort.
  + **Verwijder(𝑤)** werkt als volgt:
    - Bereken de juiste positie van het woord 𝑤 met de hashfunctie;
    - Zoek in de gelinkte lijst, waar de gevonden positie naar refereert, naar de knoop met het woord 𝑤;
    - Verwijder de gevonden knoop uit de
* Examenvraag: je hebt voor gesloten en open hashing een hashfunctie nodig:
  + waar, je hebt ze voor beide inderdaad nodig.
* Probleem open hashing:

## Keuze van hashcode en hashfunctie

Keuze van hashcode en hashfunctie:

* Om een efficiënte hashtabel op te bouwen is het niet alleen belangrijk dat de overlappingen goed worden opgevangen maar is het vooral belangrijk een efficiënte hashcode en hashfunctie te kiezen. Een goede combinatie van hashcode en hashfunctie moet aan twee voorwaarden voldoen:
  + 1. Snel te berekenen
  + 2. Zo groot mogelijke spreiding van de elementen over de verschillende posities
* Dit lijkt voor de hand liggend maar het gebeurt in de praktijk heel vaak dat er tegen de tweede voorwaarde wordt gezondigd. Dit leidt dan tot een teleurstellende performantie.

Problematische hashcode:

* Veronderstel dat we hashcode voor een string definiëren als de som van de waarden van de verschillende letters. Bv. Het woord “hashtabel” zou in dit geval de volgende waarde krijgen: 7 + 0 + 18 + 7 + 19 + 0 + 1 + 4 + 11 = 67.
* Deze hashcode heeft twee problemen:
  + 1. Waarden zijn (te) klein vb. Woorden met hoogstens 10 karakters lang geeft een maximumwaarde van 25 × 10 = 250. Zelfs als we een array alloceren met 10 000 posities dan zullen enkel de
  + eerste 250 posities gebruikt kunnen worden.
  + 2. Alle permutaties van dezelfde letters worden op dezelfde hashcode afgebeeld, zo zal het Engelstalige woord “hashtable” ook hashcode 67 hebben. Dit zorgt voor onnodig veel botsingen.

Problematische tabelgrootte:

* Ook de grootte van de hashtabel, i.e. **De waarde van 𝑁 kan een grote invloed hebben** op de performantie van de hashtabel.
* Stel : alle HOGENT studenten in een hashtabel. Als sleutelwaarde gebruiken we een uniek studentennummer voor elke student. Veronderstel dat dit studentennummer als volgt is opgebouwd. Eerst komen er 4 cijfers (een volgnummer) en daarna komt het jaar waarin deze student zich voor het eerst inschreef. Volgens deze werkwijze kan er jaarlijks aan maximaal 104 = 10 000 studenten een nummer toegekend worden. We veronderstellen dat dit voldoende is aangezien er zich gemiddeld 5000 nieuwe studenten inschrijven
* We wensen gegevens bij te houden voor de laatste 10 jaar, dus van ongeveer 50 000 studenten, en we gebruiken hiervoor een hashtabel met open hashing. Wanneer de grootte van deze hashtabel gelijk aan 10 000 wordt gekozen dan zou, bij een ideale verdeling, elke lineair gelinkte lijst ongeveer 5 elementen bevatten.
* Echter, door de gekozen tabelgrootte krijgen we het volgende:
  + ℎ(1234–2019) = 2019
  + ℎ(5489–2019) = 2019
  + ℎ(0358–2019) = 2019
  + ℎ(9786–2020) = 2020.
* Dit betekent dat alle studenten die gestart zijn in hetzelfde academiejaar op dezelfde positie terechtkomen in de tabel. Al deze overlappingen worden bijgehouden in een gelinkte lijst, maar deze gelinkte lijst zal wel bijzonder lang worden ⟹ 5000 studenten
  + Hoe kunnen we dit oplossen?
  + Waarom priemgetallen gebruiken?
    - Het vermijdt botsingen want priemgetallen hebben geen delers
  + 10 007 is het priemgetal het dichtste gelegen bij 10 000
* De hashfunctie wordt nu gegeven door
* ℎ ∶ student ↦ student.hashcode (mod 10 007).
* Deze hashfunctie zorgt ervoor dat de verschillende studenten beter verdeeld worden over alle posities van de tabel. Bijvoorbeeld:
  + ℎ(1234–2019) = 3388
  + ℎ(5489–2019) = 3624
  + ℎ(0358–2019) = 9520
  + ℎ(9786–2020) = 3567.
* De studentengegevens worden op de corresponderende positie ingevuld in de tabel.

## Oefeningen

Oef 1:

**Gesloten hashing (“democratisch”):**

0 1 2 3 4

o c m d e

De tabel zit vol. Bij open hashing kunnen we nu niks meer doen.

**Open hashing (“democratisch”):**

0 🡪 a

1 🡪 r

2 🡪h 🡪 c 🡪 s 🡪 c 🡪 m \

3 🡪 i 🡪 t 🡪 d

4 🡪 o 🡪 e

# Hoofdstuk 4 Bomen:

## Boom

Motivatie:

* Er zijn veel situaties waarin informatie geordend is volgens één of andere hiërarchische structuur. Denken we bv. Maar aan
  + Bestandsystemen
  + De structuur van XML-bestanden
  + Familiestambomen
  + Organisatorische structuren …
* De abstractie die zulke situaties modelleert is een boom, een fundamenteel begrip in de informatica.

Definitie:

* Een GEWORTELDE BOOM 𝑇 is een verzameling van toppen die aan de volgende eigenschappen voldoet:
  + 1. Er is één speciale top 𝑡 die de WORTEL van de boom wordt genoemd.
  + 2. De andere toppen zijn verdeeld in 𝑚 ≥ 0 disjuncte verzamelingen 𝑇1 , … , 𝑇𝑚 die op hun beurt elk weer een GEWORTELDE BOOM zijn.
* Opmerking: Één enkele top is ook steeds een boom aangezien het toegelaten is dat het aantal verzamelingen 𝑚 nul is. Dit is het eindgeval van de recursie

Afbeelding met schets, diagram, cirkel, lijn

Automatisch gegenereerde beschrijving

Grafische voorstelling:

* Voor deze boom is de wortel 𝑛1 .
* We hebben verder: 𝑇1 = {𝑛2 , 𝑛5 , 𝑛6 }, 𝑇2 = {𝑛3 } en 𝑇3 = {𝑛4 , 𝑛7 }.

Begrippen:

* **Deelboom**
* **Kinderen**: wortels van de deelbomen (ouder)
* **BROERS**: kinderen van dezelfde ouder
* **Afstammeling** en voorouder
* **Graad**: aantal kinderen van een top;
  + Graad van een boom: maximum graad van zijn toppen
* **BLAD**: top zonder kinderen (versus interne top)

Diepte en hoogte:

* De DIEPTE van een top 𝑛 in een boom 𝑇.
  + 1. Diepte van de wortel is nul;
  + 2. Diepte van elke andere top is één meer dan de diepte van zijn ouder.
* De DIEPTE van de boom is de maximale diepte van zijn toppen.
* De hoogte van een top is de diepte van de deelboom met die top als wortel.
* De hoogte van een boom is de hoogte van zijn wortel (is dus ook de diepte van de boom!)
* Samengevat: diepte is ’afstand’ tot wortel; hoogte is ’afstand’ tot verste blad.

Afbeelding met diagram, lijn, Plan, Technische tekening

Automatisch gegenereerde beschrijvingArray-van-kinderen voorstelling:

* Datastructuur Top met een array van referenties naar kinderen.
* **De grootte van de array is graad van de boom.**
* Geheugenplaatsen = n(7) \* graad (3) = 21
* Aantal toppen – 1 = geheugenefficiëntie

Geheugengebruik (BREUK VAN MAKEN):

* Aantal referenties: 𝑛 × 𝑘
* Aantal gebruikte referenties: 𝑛 − 1 (waarom?)
* Verhouding is dus slechts
  + (𝑛 – 1 / 𝑛\*𝑘) ≈ 1 / 𝑘 Opmerking:
* We gaan er hierbij van uit dat elke array steeds een vaste grootte heeft

Afbeelding met diagram, lijn, Plan, Technische tekening

Automatisch gegenereerde beschrijvingEerste-kind-volgende-broer voorstelling (belangrijk):

* Elke Top heeft juist twee referenties: een referentie naar zijn eerste kind, en een referentie naar zijn volgende broer.
* Kind = onder de n (linkervak)
* Broer = naast de n (rechtervak)

Geheugengebruik:

* Aantal referenties: 2𝑛 (onafhankelijk van de graad van de boom 𝑘)
* Aantal gebruikte referenties: 𝑛 − 1 (waarom?)
* Verhouding wordt nu
  + (𝑛 – 1) / (2𝑛) ≈ 1/2
* En is dus onafhankelijk van de graad van de boom.
* Nadeel: bepalen/aanspreken van bv. Derde kind wordt iets moeilijker.

Alle toppen van een boom bezoeken:

* Om een boom te doorlopen in PREORDE gaat men als volgt tewerk:
  + 1. Bezoek (eerst) de wortel van de boom.
  + 2. Doorloop de deelbomen van de wortel in preorde (recursie!)
  + **Eigen wortel eerst (printen)**
* Om een boom te doorlopen in POSTORDE doet men het volgende:
  + 1. Doorloop de deelbomen van de wortel in postorde (recursie!).
  + 2. Bezoek de wortel van de boom (laatst)
  + **Eigen kinderen eerst (printen)**
* Eindgeval van de recursie: er zijn geen deelbomen (dus top is blad)

Preorde: pseudocode:

* **Invoer** Een boom 𝑇, en een visit functie.
* **Uitvoer** De visit functie is aangeroepen voor elke top van de boom.

1: function PREORDE(𝑇,visit)

2: preorderecursief(𝑇.wortel, visit) # start met de wortel

3: end function

4: function preorderecursief(𝑣, visit)

5: visit(𝑣)

6: for all 𝑤 ∈ kinderen(𝑣) do # implementatie-onafhankelijk

7: preorderecursief(𝑤, visit)

8: end for

9: end function

Oefening preorder en postorde op slide 19:

Preorde: n1, n2, n5, n6, n3, , n4, n7

Postorde: n5, n6, n2, n3, n7, n4, n1

Eenvoudige berekeningen:

* We wensen het aantal toppen van een boom te berekenen.
* We weten: #(𝑇 ) = 1 + #(𝑇1 ) + #(𝑇2 ) + ⋯ + #(𝑇𝑚 ).
* Dit kan nu eenvoudig vertaald worden in (pseudo)code.

Pseudocode: aantal toppen:

* **Invoer** Een boom 𝑇
* **Uitvoer** Het aantal toppen van de boom.

1: function aantal(𝑇)

2: return aantalrecursief(𝑇.wortel)

3: end function

4: function aantalrecursief(𝑣)

5: 𝑛 ← 1 # Top zelf niet vergeten

6: for all 𝑤 ∈ kinderen(𝑣) do

7: 𝑛 ← 𝑛 + aantalrecursief(𝑤)

8: end for

9: return 𝑛

10: end function

## Binaire boom

Afbeelding met cirkel, schets, diagram, tekening

Automatisch gegenereerde beschrijvingBinaire boom definitie:

* Een binaire boom is een verzameling toppen die
  + 1. Ofwel **leeg** is,
  + 2. Ofwel bestaat uit een **wortel** en twee disjuncte verzamelingen 𝑇𝑙 en 𝑇𝑟 , die op hun beurt ook een binaire boom zijn. We noemen 𝑇𝑙 en 𝑇𝑟 respectievelijk de **linker**- en **rechterdeelboom** van de wortel.
* Opmerkingen:
  + Binaire boom kan leeg zijn, ”gewone”gewortelde boom niet;
  + Bij binaire bomen zijn deelbomen geordend: linker- en rechterdeelboom.
  + **Graad is maximum 2**

Eigenschappen binaire bomen:

* Stelling
  + In een binaire boom is het aantal toppen met diepte 𝑘 hoogstens 2𝑘 .
* Bewĳs.
  + Door inductie op de diepte k
* Als we de diepte van een binaire boom kennen dan kunnen we grenzen opstellen voor het aantal toppen in deze boom.
  + Stelling Voor een (niet-lege) binaire boom 𝑇 met diepte 𝑑 ≥ 0 geldt dat:
    - 𝑑 + 1 ≤ #(𝑇 ) ≤ 2𝑑+1 − 1.
    - 2 n (aantan toppen, hier 3) +1 = aantal toppen

Afbeelding met schets, diagram, tekening, wit

Automatisch gegenereerde beschrijvingAfbeelding met cirkel, schets, tekening, illustratie

Automatisch gegenereerde beschrijving

Afbeelding met diagram, schets, Technische tekening, Plan

Automatisch gegenereerde beschrijvingDatastructuur binaire boom:

* We gebruiken een datastructuur Top met twee referentievelden links en rechts.
* De boom zelf is eigenlijk een referentie naar zijn wortel.

Alle toppen bezoeken: Pre- en Postorde:

* Om een binaire boom in PREORDE te doorlopen gaan we als volgt te werk:
  + 1. Bezoek de wortel van boom.
  + 2. Als de linkerdeelboom niet leeg is, doorloop de linkerdeelboom dan recursief in preorde.
  + 3. Als de rechterdeelboom niet leeg is, doorloop de rechterdeelboom dan recursief in preorde.
* POSTORDE: idem maar bezoek aan wortel laatst.

Alle toppen bezoeken: Inorde:

* Derde mogelijkheid om een binaire boom in inorde te doorlopen gaan we als volgt tewerk:
  + 1. Als de linkerdeelboom niet leeg is, doorloop de linkerdeelboom dan recursief in inorde.
  + 2. Bezoek de wortel van boom.
  + 3. Als de rechterdeelboom niet leeg is, doorloop de rechterdeelboom dan recursief in inorde.
* **Invoer** Een binaire boom 𝑇 en een visit functie.
* **Uitvoer** De visit functie is aangeroepen voor elke top van 𝑇.

1: function inorde(𝑇,visit)

2: if 𝑇 ≠ ∅ then # controleer dat boom niet leeg is

3: inorderecursief(𝑇.wortel, visit) # start met de wortel

4: end if

5: end function

6: function inorderecursief(𝑣, visit)

7: if 𝑣.links ≠ ∅ then

8: inorderecursief(𝑣.links, visit)

9: end if

10: visit(𝑣) # visit aanroepen tussen recursieve oproepen

11: if 𝑣.rechts ≠ ∅ then

12: inorderecursief(𝑣.rechts, visit)

13: end if

14: end function

Oefening slide 33:

Inorde: n4, n2, n7, n5, n8, n1, n3, n6, n9

Oefening slide 34 :

Boom : 9, 3, 1, 0, 4, **2**, 7, 6, 8, 5

2

3 8

9 0 7 5

1 4 6

Collection:

* Vaak wensen we
  + 1. Gegevens toe te voegen aan de verzameling,
  + 2. Gegevens te verwijderen uit de verzameling, en
  + 3. Te controleren of een element aanwezig is in de verzameling.
* Mogelijke datastructuren: (lineair gelinkte) lijsten, hash-tabellen, etc.
* Een andere mogelijke datastructuur is de binaire zoekboom

Totaal geordende verzameling:

* Een noodzakelijke voorwaarde om een binaire zoekboom te kunnen gebruiken is dat de labels een totaal geordende verzameling vormen. Voor elke paar (mogelijke) labels 𝑥 en 𝑦 geldt dat
* 𝑥 < 𝑦 of 𝑦 < 𝑥 of 𝑥 = 𝑦.
* Opmerking: Vaak is er met het label andere data geassocieerd. Dit wordt in de theorie niet bekeken of vermeld.

## Binaire zoekboom

Afbeelding met cirkel, schets, diagram, tekening

Automatisch gegenereerde beschrijvingBinaire zoekboom: definitie:

* Een BINAIRE ZOEKBOOM is een gelabelde binaire boom die aan een bijzondere voorwaarde, de binaire zoekboomeigenschap, voldoet. Definitie
* De BINAIRE zoekboomis de volgende: voor elke top 𝑥 van de binaire zoekboom **geldt dat alle toppen in de linkerdeelboom van 𝑥 een label hebben dat kleiner is dan het label van 𝑥, terwijl voor alle toppen in de rechterdeelboom van 𝑥 geldt dat hun label groter is dan het label van 𝑥.**

Afbeelding met diagram, lijn, schermopname, cirkel

Automatisch gegenereerde beschrijvingOpzoeken van een sleutel:

* Om een sleutel op te zoeken in een binaire zoekboom maken we gebruik van de binaire zoekboomeigenschap om die (snel) te vinden.
  + 1. Wanneer de boom leeg is, geef dan ’niet gevonden’ terug.
  + 2. Vergelijk 𝑥 met de sleutel van de wortel.
    - 2.1 Wanneer 𝑥 kleiner is dan dit label, zoek dan (recursief) in de linkerdeelboom.
    - 2.2 Wanneer 𝑥 groter is dan dit label, zoek dan (recursief) in de rechterdeelboom.
    - 2.3 Geef de wortel van de boom terug (𝑥 werd gevonden).
* Dit is dus een recursieve procedure.
* 2 Dezelfde waarden kunnen niet in een binaire zoekboom

Opzoeken van kleinste/grootste element:

* Door de binaire zoekboomeigenschap weten we waar het kleinste element zich bevindt. Waar?
* We vinden het kleinste element als volgt:
  + 1. Wanneer de linkerdeelboom van de wortel leeg is, geef dan (de sleutel van) de wortel terug.
  + 2. In het andere geval zoek je recursief naar het kleinste element van de linkerdeelboom.

Afbeelding met diagram, lijn, schermopname, cirkel

Automatisch gegenereerde beschrijvingToevoegen van een element:

* Na toevoegen van sleutel 𝑥 moet nieuwe boom nog steeds binaire zoekboom zijn!
* Toevoegen van een sleutel 𝑥 aan een (niet-lege) binaire zoekboom:
* 1. Vergelijk 𝑥 met het label van de wortel:
  + 1.1 Wanneer 𝑥 kleiner is dan het label van de wortel, voeg dan 𝑥 toe aan de linkerdeelboom wanneer die niet leeg is (recursie!). Wanneer de linkerdeelboom leeg is vervang dan de (null)-referentie naar de linkerdeelboom door de referentie naar een nieuwe top met 𝑥 als label.
  + 1.2 Wanneer 𝑥 groter is dan het label van de wortel, voeg dan 𝑥 toe aan de rechterdeelboom wanneer die niet leeg is (recursie!). Wanneer de rechterdeelboom leeg is vervang dan de (null)-referentie naar de rechterdeelboom door de referentie naar een nieuwe top met 𝑥 als label.
  + 1.3 Doe niets, want 𝑥 behoort reeds tot de boom

Verwijderen van een sleutel:

* Verwijderen is de moeilijkste bewerking!
* Het verwijderen van een sleutel 𝑥 start met het opzoeken van deze sleutel in de boom. Er kunnen zich nu drie gevallen voordien:
* 1. De sleutel 𝑥 bevindt zich in een blad.
* 2. De sleutel 𝑥 bevindt zich in een top met één kind.
* 3. De sleutel 𝑥 bevindt zich in een top met twee kinderen.

Afbeelding met cirkel, diagram, lijn, wit

Automatisch gegenereerde beschrijvingVerwijderen van een blad:

* Laat eenvoudigweg dit blad weg. (Stel de gepaste wijzer van de ouder op null).
* Verwijder sleutel 4 uit onderstaande boom.
* Dit is geen probleem want hij heeft geen kinderen

Verwijderen top met één kind:

Afbeelding met cirkel, schets, diagram, tekening

Automatisch gegenereerde beschrijvingWanneer de top 𝑛 die 𝑥 bevat slechts één kind heeft, dan kunnen we 𝑛 vervangen door dit ene kind. Anders gezegd: het nieuwe linker- of rechterkind van de ouder van 𝑛 is zijn vroegere kleinkind. De binaire zoekboomeigenschap zal in dit geval nog steeds geldig zijn. Verwijder 10 uit onderstaande boom

Afbeelding met diagram, cirkel, lijn

Automatisch gegenereerde beschrijvingVerwijderen van een top met twee kinderen:

* We herleiden dit probleem naar een eenvoudiger probleem:
* Eerst vervangen we de inhoud van de top door zijn opvolger, i.e. De inhoud van de kleinste top in zijn rechterdeelboom (voor 8 is dit 13 bv.)
* Vervolgens verwijderen we deze kleinste top van de rechterdeelboom. (Waarom is dit eenvoudiger?!)
* Verwijder de top met sleutel 3.
* Hiervoor gebruiken we sleutel 4, want 4 is kleiner dan 6 en 6 staat rechts.

Tijdscomplexiteit:

* Al deze bewerkingen hebben een tijdscomplexiteit die lineair is in de diepte (d) van de boom.
* 1. Voor ’perfect’ gebalanceerde bomen: 𝛩(𝑑) = 𝛩(lg(𝑛)).
  + Indien je elke laag volsteekt is de diept **log n (beter)**
  + Als hij heel slecht gebalanceerd is **n (slechter)**
* 2. In het slechtste geval: 𝛩(𝑑) = 𝛩(𝑛)
* 3. Gemiddelde geval (want er zijn weining bomen die slecht zullen gebalanceerd zijn) + (bij random toevoegen sleutels): 𝛩(𝑑) = 𝛩(lg(𝑛))

Prioriteitswachtrij:

* Een PRIORITEITSWACHTRIJ is een uitbreiding van de “gewone” FIFO wachtrij.
* Elementen: sleutel en waarde. Hoe kleiner de sleutel, hoe groter de prioriteit.
* Bij een prioriteitswachtrij kan men
  + 1. Het element met de kleinste sleutel opzoeken;
  + 2. Het element met de kleinste sleutel verwijderen;
  + 3. Een nieuw element toevoegen aan de prioriteitswachtrij.
* Opmerking: In een prioriteitswachtrij kan enkel het element met de kleinste sleutel efficiënt bereikt worden. M.a.w. Invoegen is flexibel, verwijderen niet.

## Biniare hoop

Binaire hoop:

* Een COMPLETE BINAIRE BOOM is een binaire boom van diepte 𝑑 waarbij het aantal toppen met diepte 𝑘 < 𝑑 maximaal (dus 2 𝑘 , zie Eigenschap 1) is. De toppen met diepte 𝑑 komen voor van “links naar rechts”.
* De ORDENINGSEIGENSCHAP VOOR BINAIRE HOPEN zegt dat de sleutel van elke top hoogstens gelijk is aan de sleutel van zijn kinderen.

Afbeelding met diagram, schets, tekening, cirkel

Automatisch gegenereerde beschrijvingBinaire hoop m.b.v. Arrays:

* Omdat de binaire boom compleet is, is het niet nodig om effectief referenties naar kinderen e.d. Te gaan bijhouden.
* We ’nummeren’ de toppen niveau per niveau, en van links naar rechts. Bv.

Verband rangnummer ouder/kind (eigenschap):

* Wanneer een top rangnummer 𝑖 heeft, dan hebben zijn linker- en rechterkind (als die bestaan) respectievelijk rangnummer 2𝑖 en 2𝑖 + 1.
* Omgekeerd geldt: wanneer een top rangnummer 𝑖 heeft (en deze top is niet de wortel van de boom), dan heeft zijn ouder rangummer floor(𝑖/2).
* We kunnen dus stellen dat:
  + Left(𝑖) = 2𝑖
  + Right(𝑖) = 2𝑖 + 1
  + Parent(𝑖) = floor(𝑖/2).

Opzoeken van element met kleinste sleutel:

* Uit de ordeningseigenschap volgt dat het kleinste element steeds in de wortel zit.
* Het opzoeken van het kleinste element is dus een triviale bewerking

Toevoegen van een element:

* 1. Creëer een nieuw element.
* 2. Voeg dit element toe op de eerste beschikbare plaats. Dit betekent dus als een nieuw blad, met rangnummer 𝑖, waarbij we er steeds voor zorgen dat het diepste niveau gevuld is van links naar rechts.
* 3. Op dit moment is het in het algemeen zo dat de ordeningseigenschap voor binaire bomen nu kan geschonden zijn tussen 𝑖 en parent(𝑖). Indien dit zo is, verwissel dan 𝑖 en parent(𝑖). Dit herstelt de ordeningseigenschap tussen 𝑖 en zijn ouder. Eventueel is nu de ordeningseigenschap tussen parent(𝑖) en parent(parent(𝑖)) geschonden. Indien dit zo is wissel dan beide elementen. Ga zo verder tot de binaire hoop is hersteld.

Verwijderen kleinste element:

* Wanneer we het kleinste element verwijderen, dan hebben we een ’gat’ in de wortel. We vullen dit op met het laatste element. Dan moet de ordeningseigenschap hersteld worden:
* 1. Verwissel de wortel met het meest rechtse blad met de grootste diepte.
* 2. Verwijder het meest rechtse blad; de binaire hoop heeft nu een element minder.
* 3. Indien de ordeningseigenschap geschonden is in de wortel, herstel deze dan door de wortel en zijn kleinste kind 𝑖 van plaats te verwisselen. Indien de ordeningseigenschap nu geschonden is in 𝑖, herstel ze dan door 𝑖 te verwisselen met de kleinste van zijn kinderen. Ga zo verder tot de binaire hoop hersteld is

Tijdscomplexiteit:

* 1. Opzoeken van het kleinste element gebeurt in constante tijd.
* 2. Toevoegen en verwijderen neemt tijd 𝛩(lg 𝑛). (Waa

Oefeningen:

**Oef 1, geef de binaire zoekboom (4, 7, 5, 8, 11, 3, 2, 9, 10, 6):**

4

3 7

2 **5** 8

**6** 11

9

10

**Oef 2: Veronderstel dat men een binaire zoekboom opbouwt door sleutels één voor één toe te voegen aan een initieel lege boom.**

2.1 Geef een rij van lengte 7 die een binaire zoekboom van minimale diepte oplevert:

Rij = [4, 2, 6, 1, 3, 5, 7]

4

**2** 6

**1** **3** 5 7

2.2 Voor lengte 15 beginnen we bij (15 / 2) = 7.5 🡪 8

**2.3 voor lengte 5:**

1 5

2 4

3 3

4 2

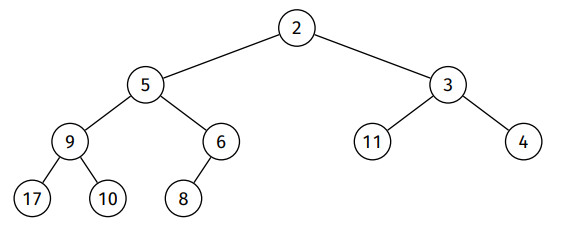
5 1

Je krijgt een slecht gebalanceerde boom als je de sleutelwaarden die je moet toevoegen in volgorde zet.

## Binaire hoop

Prioriteitswachtrij:

* Een PRIORITEITSWACHTRIJ is een uitbreiding van de “gewone” FIFO wachtrij.
* Elementen: sleutel en waarde.
* **Hoe kleiner de sleutel, hoe groter de prioriteit.**
* Bij een prioriteitswachtrij kan men
  + 1. het element met de kleinste sleutel opzoeken;
  + 2. het element met de kleinste sleutel verwijderen;
  + 3. een nieuw element toevoegen aan de prioriteitswachtrij.
* Opmerking: In een prioriteitswachtrij kan enkel het element met de kleinste sleutel efficiënt bereikt worden. **M.a.w. invoegen is flexibel, verwijderen niet.**
* Onder een ouder moeten waarden die groter (dus 5 heeft als kinderen 10 en 13 bv)
  + De parent is altijd kleiner dan zijn kinderen
* Je begint met vullen helemaal links en gaat zo door naar rechts en naar de andere broers van rechts)

Binaire hoop:

* Een Complete Binaire Boom is een binaire boom van diepte 𝑑 waarbij het aantal toppen met diepte 𝑘 < 𝑑 maximaal (dus 2 𝑘 , zie Eigenschap 1) is. De toppen met diepte 𝑑 komen voor van “links naar rechts”.
  + **Je boom is goed gevuld**, enkel de onderste is niet volledig gevuld, al de rest wel.
* De Ordeningseigenschap voor Binaire Hopen zegt dat de sleutel van elke top hoogstens gelijk is aan de sleutel van zijn kinderen.

Afbeelding met diagram, schets, tekening, lijn

Automatisch gegenereerde beschrijvingBinaire hoop m.b.v. arrays:

* Omdat de binaire boom compleet is, is het niet nodig om effectief referenties naar kinderen e.d. te gaan bijhouden.
* We ’nummeren’ de toppen niveau per niveau, en van links naar rechts. Bv.
* Dit is een complete binaire boom, want
  + Diepte 4 is niet volledig gevuld
  + Diepte 1 – 3 is wel volledig gevuld

Verband rangnummer ouder/kind

* Wanneer een top rangnummer 𝑖 heeft, dan hebben zijn linker- en rechterkind (als die bestaan) respectievelijk rangnummer **2𝑖 en 2𝑖 + 1. Wanneer we starten op 1**
* Omgekeerd geldt: wanneer een top rangnummer 𝑖 heeft (en deze top is niet de wortel van de boom), dan heeft zijn ouder rangummer floor(𝑖/2). We kunnen dus stellen dat:
  + left(𝑖) = 2𝑖
  + right(𝑖) = 2𝑖 + 1
  + parent(𝑖) = floor(𝑖/2).

Opzoeken van element met kleinste sleutel

* Uit de ordeningseigenschap volgt dat het kleinste element steeds in de wortel zit.
* Het opzoeken van het kleinste element is dus een triviale bewerking.
* Kleinste sleutel = top ofwel 1

Toevoegen van een element

* Creëer een nieuw element.
* Voeg dit element toe op de eerste beschikbare plaats. Dit betekent dus als een nieuw blad, met rangnummer 𝑖, waarbij we er steeds voor zorgen dat het diepste niveau gevuld is van links naar rechts.
* Op dit moment is het in het algemeen zo dat de ordeningseigenschap voor binaire bomen nu kan geschonden zijn tussen 𝑖 en parent(𝑖). Indien dit zo is, verwissel dan 𝑖 en parent(𝑖). Dit herstelt de ordeningseigenschap tussen 𝑖 en zijn ouder. Eventueel is nu de ordeningseigenschap tussen parent(𝑖) en parent(parent(𝑖)) geschonden. Indien dit zo is wissel dan beide elementen. Ga zo verder tot de binaire hoop is hersteld

Hoeveel verwisselingen kunnen maximaal nodig zijn bij het Omhoog Bubbelen?

Het resultaat van de diepte tot aan de hoogte van de boom.

Verwijderen kleinste element

* Wanneer we het kleinste element verwijderen, dan hebben we een ’gat’ in de wortel. We vullen dit op met het laatste element. Dan moet de ordeningseigenschap hersteld worden:
* Verwissel de wortel met het meest rechtse blad met de grootste diepte.
* Verwijder het meest rechtse blad; de binaire hoop heeft nu een element minder.
* Indien de ordeningseigenschap geschonden is in de wortel, herstel deze dan door de wortel en zijn kleinste kind 𝑖 van plaats te verwisselen. Indien de ordeningseigenschap nu geschonden is in 𝑖, herstel ze dan door 𝑖 te verwisselen met de kleinste van zijn kinderen. Ga zo verder tot de binaire hoop hersteld is.
* In het voorbeeld komt element 3 op plaats 2 en 4 komt op plaats 3. Want 3 is het kleinste kind van 2 en bij plaats 3 exact hetzelfde

Tijdscomplexiteit

* Opzoeken van het kleinste element gebeurt in constante tijd.
* Toevoegen en verwijderen neemt tijd 𝛩(lg 𝑛). (Waarom?)

## Oefeningen:

**Start met een lege binaire hoop. Voeg achtereenvolgens de volgende elementen toe aan de binaire hoop: 11, 13, 1, 15, 6, 5, 9, 16, 3, 10, 7, 4, 12, 14, 2. Teken de resulterende hoop na elke toevoeging.**

Een binaire hoop met 5 element waarbij het kleinste element helemaal beneden zit:

Alle sleutels hebben dezelfde waarde

Met 1 element is dit:

1 element

# Hoofdstuk 5: Grafen

Motivatie:

* In heel wat praktische situaties heeft men te maken met de situatie waarin ’objecten’ verbonden zijn door een bepaalde relatie:
* Steden zijn met elkaar verbonden m.b.v. Wegen; kost: de afstand in kilometer,
* Computers zijn verbonden m.b.v. Netwerkkabels; kost: de communicatiesnelheid van de verbinding,
* Luchthavens zijn met elkaar verbonden door directe vluchten; kost: de duur van de rechtstreekse vlucht.

Definitie:

* Een GRAAF 𝐺 bestaat uit een verzameling knopen 𝑉 (Vertex), en een verzameling bogen 𝐸. Elke boog verbindt twee knopen, en we noteren 𝑒 = (𝑣, 𝑤). De graaf 𝐺 wordt genoteerd als het koppel (𝑉, 𝐸), dus 𝐺 = (𝑉, 𝐸).
* Als (𝑣, 𝑤) ∈ 𝐸 (bestaat het), dan zijn 𝑣 en 𝑤 adjacent; 𝑣 en 𝑤 zijn incident (liggen op de boog) met 𝑒 (en omgekeerd)
* De buren van een knoop 𝑣 zijn alle knopen 𝑤 die adjacent zijn met 𝑣;
* Aantal buren van een knoop: **graad** van die knoop (aantal buren van de knop van v naar w)
* #𝑉, wordt de ORDE van de graaf genoemd; notatie 𝑛
* #𝐸, noemt men de grootte; notatie 𝑚

Afbeelding met diagram, lijn, cirkel, ontwerp

Automatisch gegenereerde beschrijvingGerichte versus ongerichte graaf:

* Wanneer de boogparen niet geordend zijn dan spreekt men van een ongerichte graaf.
* Wanneer de boogparen wel geordend zijn dan heeft men een gerichte graaf. In een gerichte graaf heeft een boog (𝑣, 𝑤) een STAART 𝑣 en een KOP 𝑤.
* Gewogen graaf: associeer getal met de bogen
* Bogenverzameling:
  + {(a, b), (a, c), (c,d), (c,e), (e, f), (e, d), (b, d)}
* Voorbeelden
  + **Vriendschapsgraaf Facebook: ongericht (geen pijl)**
  + **Volgersgraaf Twitter, vriendschap op isntagram: gericht (wel pijl)**

Paden en cykels:

* Definitie Een PAD in een graaf 𝐺 is een opsomming van knopen (𝑣1 , 𝑣2 , … , 𝑣𝑘 ) zodanig dat er een boog bestaat tussen 𝑣𝑖 en 𝑣𝑖+1 voor 𝑖 ∈ {1, 2, … , 𝑘 − 1}.
* De lengte van dit pad is 𝑘 − 1, zijnde het aantal bogen op dit pad.
* Opmerking: 𝑘 = 1 is toegestaan: (𝑣) is een pad van lengte nul.
* Een enkelvoudige CYKEL in een graaf is een pad waarvan de lengte strikt positief is en dat begint en eindigt in dezelfde knoop, waarbij alle knopen (behalve de start- en eindknoop) verschillend zijn en waarbij bovendien geen boog méér dan een keer wordt doorlopen.

De Adjacentiematrix:

* Afbeelding met diagram, lijn, ontwerp

  Automatisch gegenereerde beschrijvingWe veronderstellen dat de knopen van de graaf 𝐺 = (𝑉, 𝐸) genummerd zijn van 1 t.e.m. 𝑛. Wanneer we te maken hebben met een (ongewogen) graaf dan kunnen we deze voorstellen door een adjacentiematrix 𝐴. Voor deze adjacentiematrix geldt:
* 𝐴𝑖,𝑗 =
  + { 1 als (𝑖, 𝑗) ∈ 𝐸
  + 0 anders.
* Opmerking: Voor een ongerichte graaf is de adjacentiematrix steeds symmetrisch, i.e. 𝐴 𝑇 is steeds gelijk aan 𝐴.

Geheugen- en tijdsgebruik:

* Geheugenruimte steeds 𝛩(𝑛2 ).
* Maximaal aantal bogen in een graaf is 𝛩(𝑛2 ) (waarom?)
* In een IJLE graaf wordt zo heel wat geheugen verspild.
* Bepalen of twee knopen adjacent zijn: 𝛩(1).
* Alle buren bepalen/overlopen: 𝛩(𝑛), onafhankelijk van de graad van de knoop

Voorbeeld gelabelde en gewogen graaf:

Afbeelding met tekst, schermopname, Lettertype, nummer

Automatisch gegenereerde beschrijving

Afbeelding met diagram, lijn

Automatisch gegenereerde beschrijvingAdjacentielijst-voorstelling:

De adjacentielijst-voorstelling van een graaf 𝐺 bestaat uit een array van toppen, genummerd 1 t.e.m. 𝑛. Op de plaats 𝑖 van deze array worden, in een lineair gelinkte lijst, de buren van top 𝑖 bijgehouden. Voor de voorbeeldgraaf:

Geheugen- en tijdsgebruik:

* De adjacentielijst-voorstelling gebruikt 𝛩(𝑛 + 𝑚) geheugenruimte.
* Het bepalen of twee knopen adjacent zijn kan nu niet langer in constante tijd gebeuren. Om te weten of 𝑖 en 𝑗 adjacent zijn moeten we immers de gelinkte lijst horend bij 𝑖 overlopen om na te gaan of 𝑗 in deze lijst aanwezig is.
* Als we alle buren van een knoop 𝑖 willen overlopen dan gebeurt dit nu in een tijd die lineair is in het aantal buren van de knoop 𝑖. Dit is theoretisch de best mogelijke uitvoeringstijd.

Implementatie m.b.v. “moderne” collections:

* Graaf geeft relaties/verbanden tussen knopen.
* Map, of dictionary datastructuren zijn uitstekend geschikt voor het aangeven van dergelijke relaties. Bv. Beeld elke knoop (of zijn label) af op de verzameling van zijn buren (bij ongewogen graaf) of op een verzameling van tupels (bij gewogen graaf).

Zoeken in Grafen: Inleiding:

* Veronderstel dat een graaf 𝐺 = (𝑉, 𝐸) gegeven is. Dan kunnen we geïnteresseerd zijn om algoritmes te vinden die de volgende vragen kunnen beantwoorden:
* 1. Welke knopen kunnen we bereiken vanuit een knoop 𝑣?
* 2. Bestaat er een pad van 𝑣 naar een specifieke knoop 𝑤?
* 3. Wat is het kortste pad van 𝑣 naar 𝑤?

Generiek zoeken:

* Doel van het algoritme: Start vanaf knoop 𝑠; vind alle knopen 𝑣 waarvoor er een pad is van 𝑠 naar 𝑣.
* Idee: initieel ontdekte gebied = knoop 𝑠
* Breid in elke stap het ontdekte gebied uit door een boog (𝑢, 𝑣) te volgen die de “grens” oversteekt.
* Op deze manier voeg je 𝑣 toe aan het ontdekte gebied.
* Stop wanneer je geen bogen meer kan volgen, i.e. Wanneer er geen bogen meer zijn die de grens tussen ontdekt en onontdekt gebied oversteken.

Generiek zoeken: pseudocode:

* **Invoer** Een gerichte of ongerichte graaf 𝐺 = (𝑉, 𝐸) met orde 𝑛 > 0. Een knoop 𝑠 waarvan het zoeken vertrekt. De knopen zijn genummerd van 1 tot 𝑛, i.e. 𝑉 = {1, 2, … , 𝑛}.
* **Uitvoer** Een array 𝐷 met 𝐷[𝑣] = true als en slechts als er een pad bestaat van 𝑠 naar 𝑣.

1: function zoekgeneriek(𝐺, 𝑠)

2: 𝐷 ← [false, false, … , false] # 𝑛 keer false

3: 𝐷[𝑠] ← true # markeer 𝑠

4: while ∃(𝑢, 𝑣) ∶ 𝐷[𝑢] = true ∧ 𝐷[𝑣] = false do

5: kies een boog (𝑢, 𝑣) met 𝐷[𝑢] = true ∧ 𝐷[𝑣] = false

6: 𝐷[𝑣] ← true # markeer 𝑣

7: end while

8: return 𝐷

Generiek Zoeken: eigenschap

Wanneer het algoritme voor generiek zoeken eindigt dan geldt voor elke knoop 𝑣 van 𝐺 dat 𝑣 gemarkeerd is als “ontdekt” (i.e. 𝐷[𝑣] = true) als en slechts als er een pad bestaat van 𝑠 naar 𝑣 in 𝐺.

Breedte-Eerst Zoeken:

* Idee: bezoek de knopen in “lagen”.
* Eerst 𝑠 zelf, dan de knopen die één boog verwijderd zijn van 𝑠, dan de knopen die twee bogen verwijderd zijn van 𝑠, enzovoort.
* Gebruikte datastructuur: wachtrij (FIFO)

Breedte-Eerst Zoeken Pseudocode:

* Invoer Een gerichte of ongerichte graaf 𝐺 = (𝑉, 𝐸) met orde 𝑛 > 0. Een startknoop 𝑠; 𝑉 = {1, 2, … , 𝑛}.
* Uitvoer Een array 𝐷 met 𝐷[𝑣] = true asa ∃ pad van 𝑠 naar 𝑣.

1: function BREEDTEEERST(𝐺, 𝑠)

2: 𝐷 ← [false, false, … , false] # 𝑛 keer false

3: 𝐷[𝑠] ← true # markeer 𝑠

4: 𝑄.init() # wachtrij van knopen

5: 𝑄.enqueue(𝑠)

6: while 𝑄 ≠ ∅ do

7: 𝑣 ← 𝑄.dequeue()

8: for all 𝑤 ∈ buren(𝑣) do

9: if 𝐷[𝑤] = false then # 𝑤 nog niet ontdekt

10: 𝐷[𝑤] ← true

11: 𝑄.enqueue(𝑤)

12: return 𝐷

Breedte-Eerst Zoeken: uitvoeringstijd:

Wanneer een adjacentielijst-voorstelling gebruikt wordt voor een graaf 𝐺, dan is de uitvoeringstijd 𝑇 (𝑛, 𝑚) van Algoritme 5.2 van de grootte-orde 𝛩(𝑛 + 𝑚).

Diepte-Eerst Zoeken:

* Idee: zo snel mogelijk zo diep mogelijk in de graaf. (zie: http://xkcd.com/761/ )
* We bezoeken steeds de meest recent ontdekte knoop.
* We gebruiken m.a.w. Een LIFO structuur (stapel) i.p.v. Een wachtrij.
* We gebruiken de impliciete call-stack

Diepte-Eerst Zoeken:

* Invoer Een gerichte of ongerichte graaf 𝐺 = (𝑉, 𝐸) met orde 𝑛 > 0. Een startknoop 𝑠; 𝑉 = {1, 2, … , 𝑛}.
* Uitvoer Een array 𝐷 met 𝐷[𝑣] = true asa ∃ pad van 𝑠 naar 𝑣. 1: function DIEPTEEERST(𝐺, 𝑠)

1: function DIEPTEEERST(𝐺, 𝑠)

2: 𝐷 ← [false, false, … , false] # 𝑛 keer false

3: diepteeerstrecursief(𝐺, 𝑠, 𝐷)

4: return 𝐷

5: end function

6: function diepteeerstrecursief(𝐺, 𝑣, 𝐷)

Afbeelding met schets, tekening, diagram, cirkel

Automatisch gegenereerde beschrijving7: 𝐷[𝑣] ← true # markeer 𝑣

8: for all 𝑤 ∈ buren(𝑣) do

9: if 𝐷[𝑤] = false then # 𝑤 nog niet ontdekt

10: diepteeerstrecursief(𝐺, 𝑤, 𝐷)

11: end if

12: end for

13: end function+

Topologisch Sorteren

* Precedentiegraaf van software modules. 10 1 3 2 5 4 7 6 9 8
* In welke volgorde kunnen de modules gecompileerd worden?
  + 1, 7, 2, 9, 4, 6, 3, 5, 8, 10

Topologische Sortering: definitie

* Een topologische sortering van een gerichte graaf 𝐺 kent aan elke knoop 𝑣 een verschillend rangnummer 𝑓(𝑣) toe van 1 t.e.m. 𝑛 zodanig dat de volgende eigenschap geldt:
* Afbeelding met schets, tekening, cirkel, diagram

  Automatisch gegenereerde beschrijving∀(𝑢, 𝑣) ∈ 𝐸 ∶ 𝑓(𝑢) < 𝑓(𝑣), (1)
* M.a.w. Als (𝑢, 𝑣) een boog is in de graaf dan is het rangnummer van de kop 𝑣 groter dan het rangnummer van de staart 𝑢.
* Wanneer we alle knopen op een rechte lijn tekenen, gesorteerd volgens het rangnummer van hun topologische sortering, dan zullen alle bogen vooruit wijzen.

Eigenschap Topologisch Sorteren:

* Eigenschap
  + Wanneer een gerichte graaf 𝐺 **geen enkelvoudige cykels heeft**, dan bestaat er een topologische sortering van 𝐺. (als er een cykel is heeft iedereen een buur, in een graaf zonder cykel is er altijd iemand zonder buur, meestal de laatste)
* Bewĳs.
  + Een gerichte graaf 𝐺 zonder cykels heeft een knoop zonder buren.
  + De knoop zonder buren is een goede kandidaat om rangnummer 𝑛 te krijgen.
  + Eenvoudig algoritme om topologische sortering te vinden.

Topologisch Sorteren: Algoritme

* We kunnen **diepte-eerst** zoeken aanpassen om een topologische sortering te geven.
* Idee: DFS zoekt vanuit elke knoop 𝑣 alle knopen 𝑤 waarvoor er een pad van 𝑣 naar 𝑤 bestaat. In een topologische sortering moet 𝑣 dus vóór 𝑤 komen.
* Hou een lijst 𝑆 bij; wanneer 𝑣 is ’afgewerkt’ met DFS, voeg dan 𝑣 vooraan toe.
* DFS moet eventueel meerdere malen worden opgeroepen!
* Je zoekt naar elementen zonder buren en zet ze volledig achteraan

Topologisch Sorteren: Ontdekken Cykel

* Initieel alle knopen op 0 (onontdekt)
* Volledig afgewerkte knopen op 2 (afgewerkt) (deze knopen hebben al een plaats in de sortering)
* Knopen waarvoor we nog zoeken naar opvolgers op 1 (bezig)
* Stel we zijn bezig met de buren van knoop 𝑣 (dus 𝑣 op 1). We vinden buur 𝑤 met toestand ’bezig’. Dit betekent dat er een pad is van 𝑤 naar 𝑣; samen met de boog (𝑣, 𝑤) wordt dit een cykel!

Topologisch Sorteren: pseudocode:

* **Invoer** Een gerichte graaf 𝐺 = (𝑉, 𝐸) met orde 𝑛 > 0. De knopen zijn genummerd van 1 tot 𝑛, i.e. 𝑉 = {1, 2, … , 𝑛}.
* **Uitvoer** Een topologische sortering van 𝐺 **indien mogelijk**, false anders.

1: function sorteertopologisch(𝐺)

2: global cycledetected ← false # globale variabele

3: 𝐷 ← [0, 0, … , 0] # 𝑛 keer 0

4: 𝑆 ← ∅ # 𝑆 is lege lijst

5: for all 𝑠 ∈ 𝑉 do

6: if 𝐷[𝑠] = 0 then # 𝑠 nog niet gezien

7: dfstopo(𝐺, 𝑠, 𝐷, 𝑆) # 𝑆 en 𝐷 referentieparameters

8: if cycledetected = true then # controleer op cykel

9: return false

10: end if

11: end if

12: end for

13: return S

Uitleg code met de tekening van 2 pagina’s terug:

Cycle detected wordt false

D = [0, 0, 0, 0, 0, 0, 0, 0, 0 , 0]

S = []

Als D[1] = 0, roep DFSTOPO op 1

D = [0, 0, 0, 0, 0, 0, 0, 0, 0 , 0] wordt D = [1, 0, 0, 0, 0, 0, 0, 0, 0 , 0]

Als D[2] = 0, roep DFSTOPO op 2

D = [1, 0, 0, 0, 0, 0, 0, 0, 0 , 0] = D = [1, 1, 0, 0, 0, 0, 0, 0, 0 , 0]

Als D[4] = 0, roep DFSTOPO op 4

D = [1, 1, 0, 0, 0, 0, 0, 0, 0 , 0] = D = [1, 1, 0, 1, 0, 0, 0, 0, 0 , 0]

Als D[6] = 0, roep DFSTOPO op 6

D = [1, 1, 0, 1, 0, 0, 0, 0, 0 , 0] = D = [1, 1, 0, 1, 0, 1, 0, 0, 0 , 0]

Pseudocode vervolg:

1: function DFSTOPO(𝐺, 𝑣, 𝐷, 𝑆)

2: 𝐷[𝑣] ← 1 # markeer 𝑣 als ’bezig’

3: for all 𝑤 ∈ buren(𝑣) do

4: if 𝐷[𝑤] = 0 ∧ cycledetected = false then # 𝑤 nog niet ontdekt

5: dfstopo(𝐺, 𝑤, 𝐷, 𝑆)

6: else if 𝐷[𝑤] = 1 then # cykel ontdekt 𝑤 ❀ 𝑣 → 𝑤

7: cycledetected ← true

8: end if

9: end for

10: 𝐷[𝑣] ← 2 # markeer 𝑣 als ’voltooid’

11: voeg 𝑣 vooraan toe aan 𝑆 # ken rangnummer toe aan 𝑣

12: end function

Oefeningen:

* Vind alle mogelijke topologische sorteringen van de graaf in Figuur 5.10.
* Vind de compilatievolgorde van de modules in Figuur 5.9 wanneer de labels in dalende volgorde worden doorlopen.
* Veronderstel nu dat er in de graaf van Figuur 5.9 een extra boog (8, 6) wordt toegevoegd. Pas nu het algoritme voor topologisch sorteren toe.
  + 1, 2, 4, 6, 3, 5, 8 allemaal op 1, daarna gaat 8 direct naar 6. Cycledetected <- true

Kortste Pad in een Ongewogen Graaf:

* We wensen te weten hoeveel stappen (bogen) men nodig heeft om, startend vanaf een top 𝑠, een andere top 𝑣 te bereiken.
* We zoeken m.a.w. De lengte van een kortste pad.
* Breedte-Eerst Zoeken overloopt de graaf ’laag per laag’. Kleine aanpassing nodig om kortste pad bij te houden.

Pseudocode:

* Invoer Een gerichte of ongerichte ongewogen graaf 𝐺 = (𝑉, 𝐸). Een startknoop 𝑠; 𝑉 = {1, 2, … , 𝑛}.
* Uitvoer De array 𝐷 met 𝐷[𝑣] de kortste afstand van 𝑠 tot 𝑣; als 𝐷[𝑣] = ∞ dan is er geen pad van 𝑠 naar 𝑣.

1: function kortstepadongewogen(𝐺, 𝑠)

2: 𝐷 ← [∞, ∞, … , ∞] # 𝑛 keer ∞

3: 𝐷[𝑠] ← 0 # kortste pad van 𝑠 naar zichzelf heeft lengte 0

4: 𝑄.init() # wachtrij van knopen

5: 𝑄.enqueue(𝑠)

6: while 𝑄 ≠ ∅ do

7: 𝑣 ← 𝑄.dequeue()

8: for all 𝑤 ∈ buren(𝑣) do

9: if 𝐷[𝑤] = ∞ then # 𝑤 nog niet ontdekt

10: 𝐷[𝑤] ← 𝐷[𝑣] + 1

11: 𝑄.enqueue(𝑤)

12: return 𝐷

13: end function

Pseudocode voor een gewogen ongerichte graaf:

1: function kortstepadgewogen(𝐺, 𝑠)

2: 𝐷 ← [∞, ∞, … , ∞] # 𝑛 keer ∞

3: 𝐷[𝑠] ← 0 # kortste pad van 𝑠 naar zichzelf heeft lengte 0

4: 𝑄.init() # wachtrij van knopen

5: 𝑄.enqueue(𝑠)

6: while 𝑄 ≠ ∅ do

7: 𝑣 ← 𝑄.dequeue()

8: for all 𝑤 ∈ buren(𝑣) do

9: if 𝐷[𝑤] = < D[v] then # 𝑤 nog niet ontdekt

10: 𝐷[𝑤] ← 𝐷[𝑣] + C(v, w) # kost van de boog (v, w)

11: 𝑄.enqueue(𝑤)

12: return 𝐷

13: end function

Kortste Pad in Gewogen Graaf:

* In een gewogen graaf willen we meestal pad met kleinste gewicht (en niet noodzakelijk met het minst aantal bogen).
* Naïeve aanpassing van breedte-eerst:
  + 𝐷[𝑤] ← 𝐷[𝑣] + gewicht(𝑣, 𝑤).
  + (i.p.v. 𝐷[𝑤] ← 𝐷[𝑣] + 1)

Algoritme van Dijkstra Sleutelideeën:

* Op elk moment: een verzameling 𝑆 van knopen 𝑣 waarvoor de kortste afstand van 𝑠 tot 𝑣 reeds gekend is, en een verzameling 𝑄 van knopen waarvoor de kortste afstand nog niet met zekerheid gekend is.
* Voor elke knoop 𝑣 (die tot 𝑄 behoort): 𝐷[𝑣] is de kortste afstand van een pad van 𝑠 naar 𝑣 dat enkel uit knopen van 𝑆 bestaat (behalve de laatste).
* We voegen telkens dié knoop 𝑣 van 𝑄 toe aan 𝑆 waarvoor 𝐷[𝑣] minimaal is onder alle knopen van 𝑄. Dit betekent dat voor de buren 𝑤 van 𝑣 die tot 𝑄 behoren we eventueel 𝐷[𝑤] moeten aanpassen. Het pad van 𝑠 naar 𝑣 (dat nu enkel uit knopen van 𝑆 bestaat) uitgebreid met de boog (𝑣, 𝑤) zou eventueel korter kunnen zijn dan het tot dan toe gevonden kortste pad.

Dijkstra: Pseudocode

* **Invoer** Een gewogen graaf 𝐺 = (𝑉, 𝐸) met positieve gewichten. Startknoop 𝑠; 𝑉 = {1, 2, … , 𝑛}.
* **Uitvoer** De array 𝐷 met 𝐷[𝑣] de kortste afstand van 𝑠 tot 𝑣; als 𝐷[𝑣] = ∞ dan is er geen pad van 𝑠 naar 𝑣.

1: function DIJKSTRA(𝐺, 𝑠)

2: 𝐷 ← [∞, ∞, … , ∞] # 𝑛 keer ∞

3: 𝐷[𝑠] ← 0 # kortste pad van 𝑠 naar zichzelf heeft lengte 0

4: 𝑄 ← 𝑉 # knopen waarvan kortste afstand nog niet is bepaald

5: while 𝑄 ≠ ∅ do

6: zoek 𝑣 ∈ 𝑄 waarvoor 𝐷[𝑣] minimaal is (voor knopen in 𝑄)

7: verwijder 𝑣 uit 𝑄

8: for all 𝑤 ∈ buren(𝑣) ∩ 𝑄 do

9: if 𝐷[𝑤] > 𝐷[𝑣] + gewicht(𝑣, 𝑤) then

10: 𝐷[𝑤] ← 𝐷[𝑣] + gewicht(𝑣, 𝑤) # korter pad 𝑠 → 𝑤

11: return 𝐷

12: end function

Implementatie:

* In Dijkstra moeten herhaaldelijk minima worden berekend: prioriteitswachtrij (bv. Binaire hoop)!
* Probleem: sleutels moeten worden aangepast (verminderd).
* Standaard niet mogelijk met een binaire hoop. Uitbreiding mogelijk m.b.v. Tweede array.

Oefeningen:

Vind voor de graaf in Figuur 5.4 de lengte van het kortste pad van Brugge naar alle andere steden. Voer hiertoe het algoritme van Dijkstra uit:

Q = [Brugge, Gent, Brussel, Antwerpen, Leuven, Bergen, Namen, Aarken, Lui, Hasselt, Waver]

D = []

4. Vind voor de graaf in Figuur 5.14 (de lengte van) het kortste pad van de knoop 𝑎 naar alle andere knopen. Voer hiertoe het algoritme van Dijkstra uit (en houd ook bij wat de kortste paden zijn). (100% een examenvraag)

Afbeelding met lijn, cirkel, tekening, diagram

Automatisch gegenereerde beschrijving

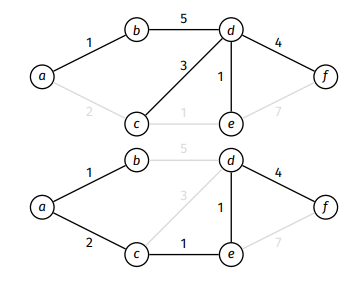
Q = [a b c d e]

P = [a c a b c]

D = [~~&~~ & & & &]

0 ~~11~~ 5 ~~11~~ 7

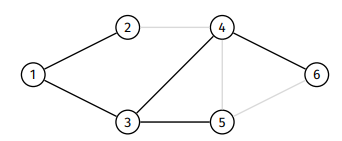
8 10

Opspannende Boom: Definitie

Een opspannende boom van een ongerichte graaf 𝐺 = (𝑉, 𝐸) is een **verzameling van bogen** 𝑇, met 𝑇 ⊆ 𝐸, zodanig dat 𝐺 ′ = (𝑉, 𝑇 ) een pad heeft tussen elke twee knopen van 𝑉, en zodanig dat 𝐺 ′ geen enkelvoudige cykels heeft.

Minimale Kost Opspannende Boom:

* Het gewicht van een boom is de som van de gewichten van zijn bogen:
* Gewicht(𝑇 ) = ∑ 𝑡∈𝑇 gewicht(𝑡). Definitie
* Een minimale KOST opspannende boom 𝑇 van de graaf 𝐺 is een opspannende boom zodanig dat voor alle (andere) opspannende bomen 𝑇 ′ van 𝐺 geldt dat
* ∑ 𝑡∈𝑇 gewicht(𝑡) ≤ ∑ 𝑡 ′∈𝑇′ gewicht(𝑡′ ).
* Hij neemt altijd de kortste paden naar welke knoop dan ook.

Algoritme van Prim:

* Het algoritme voor generiek zoeken levert reeds een opspannende boom! (Als we de bogen zouden bijhouden). Dit is bijvoorbeeld een mogelijke uitvoer van generiek zoeken: 1 2 3 4 5 6
* Essentie Prim: doe generiek zoeken maar kies steeds de goedkoopste beschikbare boog: een gulzige strategie

Algoritme van Prim: Code:

* Invoer Een ongerichte gewogen graaf 𝐺 = (𝑉, 𝐸) met orde 𝑛 > 0. De knopen zijn genummerd van 1 tot 𝑛, i.e. 𝑉 = {1, 2, … , 𝑛}.
* Uitvoer Een verzameling 𝑇 van bogen die een minimale kost opspannende boom is.

1: function prim(𝐺)

2: 𝐷 ← [false, false, … , false] # 𝑛 keer false

3: 𝐷[1] ← true # kies knoop 1 als startknoop

4: 𝑇 ← ∅ # gekozen bogen

5: while ∃(𝑢, 𝑣) ∶ 𝐷[𝑢] = true ∧ 𝐷[𝑣] = false do

6: kies (𝑢, 𝑣) met 𝐷[𝑢] = true ∧ 𝐷[𝑣] = false met minimaal gewicht

7: 𝐷[𝑣] ← true

8: 𝑇 ← 𝑇 ∪ {(𝑢, 𝑣)} # Voeg boog (𝑢, 𝑣) toe aan boom

9: end while

10: return 𝑇

11: end function

Afbeelding met lijn, diagram, cirkel

Automatisch gegenereerde beschrijvingVoorbeeld prim:

Hier ga je van: 1 => 2 => 3 => 5 => 4 => 6

Algoritme van Kruskal:

* Alternatief algoritme om minimale kost opspannende boom te bepalen.
* Eveneens een gulzig algoritme.
* We kiezen steeds de goedkoopste boog. Bogen zijn niet steeds met elkaar verbonden.
* Boog wordt enkel gekozen indien die geen cykel veroorzaakt.

Oefeningen:

**1. Vind een minimale kost opspannende boom m.b.v. het algoritme van Prim voor de graaf in Figuur 5.4. Neem als startknoop “Brugge”:**

* Je gaat van: Brugge => Gent (57) => Brussel (55) => Leuven (30) => Waver (25) => Namen (38)
* Koppels (Brugge, Gent), (Gent, Brussel), (Brussel, Leuven), (Leuven, Waver), (Waver, Namen), (Brussel, Antwerpen), (Leuven Hasselt), (Luik Hasselt), (Brussel Bergen), (Luik Aarlen)

**2. Vind een minimale kost opspannende boom m.b.v. het algoritme van Kruskal voor de graaf in Figuur 5.4:**

Je gaat van: Brugge =>

Het Handelsreizigersprobleem Definitie:

Het HANDELSREIZIGINGSPROBLEEM is het volgende: gegeven een complete¹ gewogen ongerichte graaf 𝐺 met niet-negatieve gewichten, vind dan een ordening van de knopen zodanig dat elke knoop juist éénmaal wordt bezocht (behalve de start- en eindknoop die samenvallen) en zodanig dat de som van de gewichten van de gekozen bogen minimaal is.

Moeilijkheidsgraad:

* Voor alle vorige problemen hebben we steeds een efficiënt (polynomiaal) algoritme kunnen geven.
* Het handelsreizigersprobleem is echter een NP-compleet probleem, wat hoogstwaarschijnlijk betekent dat er geen polynomiaal algoritme bestaat dat alle gevallen correct kan oplossen.
* Triviaal algoritme: probeer alle mogelijkheden en selecteer de beste. Aantal mogelijkheden is echter (𝑛 − 1)! (Je kan de eerste stad steeds als ’vast’ beschouwen.)
* Men gebruikt dan ook vaak benaderende algoritmen.

Afbeelding met diagram, lijn

Automatisch gegenereerde beschrijvingSteden op een grid:

Afstand van a naar d en b => 2

Afstand van e naar g => 12 + 12 = wortel 2

Driehoeksongelijkheid:

* Voor de vorige graaf is het steeds korter om rechtstreeks van 𝑣 naar 𝑤 te gaan dan om een omweg te maken via 𝑢. De graaf voldoet aan de driehoeksongelijkheid.
* Een graaf 𝐺 voldoet aan de driehoeksongelijkheid wanneer voor alle knopen 𝑢, 𝑣 en 𝑤 geldt dat
* Gewicht(𝑣, 𝑤) ≤ gewicht(𝑣, 𝑢) + gewicht(𝑢, 𝑤).

Benaderingsalgoritme voor Handelsreizigersprobleem:

* Bereken een minimale kost opspannende boom 𝑇 voor de graaf.
* Kies willekeurig een wortel 𝑟 van deze boom.
* Geef de cykel terug die correspondeert met het in preorde doorlopen van deze boom.
* Wanneer de graaf 𝐺 aan de driehoeksongelijkheid voldoet dan is de gevonden oplossing hoogstens tweemaal zolang als de optimale oplossing.

Kleine herhaling via de les voor tijdscomplexiteit in voorbije hoofdstukken:

* Nlogn = een gesorteerde array opzoeken
* Opzoeken in een array: n
* Toevoegen in een array: n
* Verwijderen in een array: n
* GelinkteLijst opzoeken/verwijderen = constant
* Voordeel hashtabellen: opzoeken en toevoegen = constant

# Hoofdstuk 6: Zoekalgoritmes

## Inleiding

De omgeving:

* Agent bevindt zich in bepaalde beginpositie en moet naar een eindpositie (doel).
* Moet **toestand bereiken** waar één of andere **voorwaarde** voldaan is.
* Agent (de persoon) kan de acties op voorhand bepalen!
  + Eenpersoons
  + Compleet observeerbaar
  + Deterministisch
  + Statisch
  + Discreet

Definitie Zoekprobleem:

* Een zoekprobleem bestaat uit de volgende elementen:
* Een Toestandsruimte 𝑆 die alle mogelijke toestanden bevat.
* Een verzameling van mogelijke acties 𝐴.
* Een Transitie model:
  + 𝑇 ∶ (𝑆, 𝐴) → 𝑆 ∶ (𝑠, 𝑎) ↦ 𝑠′ .
* Hierbij wordt 𝑠 ′ een opvolger van 𝑠 genoemd.
* Het uitvoeren van een actie op een bepaalde toestand heeft meestal een bepaalde kost:
  + 𝑐 ∶ (𝑆, 𝐴, 𝑆) → ℝ ∶ (𝑠, 𝑎, 𝑠′ ) ↦ 𝑐(𝑠, 𝑎, 𝑠′ ).
* Opmerking: deze definitie van kost kan ook gebruikt worden in stochastische omgevingen.

Definitie Zoekprobleem (Vervolg):

* Een initiële toestand 𝑠0 ∈ **𝑆 (Startknoop);** dit is de toestand van waaruit het zoeken zal vertrekken.
* Een DOELTEST. Dit is een functie die voor elke toestand 𝑠 aangeeft of het doel bereikt is of niet. Een toestand waarvoor de doeltest voldaan is noemen we een doelstand

Toestandsruimtegraaf:

* In de toestandsruimtegraaf stellen de knopen de toestanden voor.
* Elke toestand komt m.a.w. juist één keer voor.
* Twee knopen zijn adjacent als de ene toestand de opvolger is van de andere toestand.
* De gewichten van de bogen = de kost van de **acties (bogen)**.
* Opletten: toestandsruimtegraaf wordt snel zeer groot!

Afbeelding met schermopname, plein, nummer, Rechthoek

Automatisch gegenereerde beschrijvingDe 8-puzzel:

* De 8-puzzel is een typisch voorbeeld van een zoekprobleem:
* Toestandsruimte = alle mogelijke configuraties.
* Acties: nl. Boven, Onder, Links en Rechts.
  + Beschreven in termen van lege vakje.
* Transitiemodel: zoals in de fysica van het probleem.
  + Actie Rechts wisselt 8 en blanco van plaats om in linkse puzzel op voorgaande slide.
* De kost van elke actie is één.
* Doeltest: verifiëren dat bepaalde vastgelegde configuratie (bv. deze rechts op de figuur) werd bereikt.

Wat zijn hier de componenten?

* 𝑆 (Startpunten)?
  + Array [4, 1, 2, 0, 8, 7, 6, 3, 5]
* Acties (bogen definiëren)?
  + Verplaatsing van het lege vakje (Boven, Links, Onder, Rechts)
  + Elke kost is 1
* Transitiemodel?
  + Je moet binnen de grid blijven
* Initiële toestand?
* Doeltest?

8/15/24/…-puzzel:

* Aantal toestanden wordt snel zeer groot wanneer men grotere puzzels beschouwt.
* 8-puzzel heeft 9! = 362880 toestanden. Doenbaar.
* 15-puzzel heeft 16! ≈ 2.1 × 1013 toestanden. Niet doenbaar.
* 24-puzzel heeft 25! ≈ 1.6 × 1025 toestanden. Niet doenbaar.
* Afbeelding met schermopname, patroon, plein

  Automatisch gegenereerde beschrijvingVoor grotere puzzels is het zeker niet mogelijk om de volledige toestandsruimtegraaf op te bouwen.

8-koninginnenprobleem

Een mogelijke oplossing van het 8-koninginnenprobleem.

Eerste formulering als zoekprobleem:

* 𝑆 bevat alle configuraties met hoogstens 8 koninginnen
* Transitiemodel: voeg koningin toe aan leeg vakje wanneer minder dan 8 koninginnen op het bord.
* Kost van acties is irrelevant en wordt = 0 genomen.
* Initiële toestand = leeg bord
* Doeltest: controleer dat er 8 koninginnen op het bord staan en dat er geen paar elkaar aanvalt.
  + Dit is dus niet zomaar een opsomming van doeltoestanden!

Hoeveel toestanden kan je beschouwen met deze formulering?

( 64 / 0 ) + ( 64 / 1 ) + ⋯ + ( 64 / 8 ) ≈ 5.13 × 109 , wat met 8 bytes per toestand reeds 40 GB aan hoofdgeheugen vraagt!

Les: 0 koningen => 1 mogelijke positie, 1 koningin => 64 | 2 => 64 \* 63 | 3 => 64 \* 63 \* 62

Tweede formulering als zoekprobleem:

* Toestandsruimte bevat configuraties met 𝑛 koninginnen op het bord, één in elk van de 𝑛 eerste kolommen. Geen paar koninginnen mag elkaar aanvallen.
* Acties: voeg koningin toe aan eerste vrije kolom op een niet bedreigd vakje.
* Transitiemodel: zoals men verwacht.
* Kost van acties: nog steeds nul.
* Initiële toestand en doeltest zoals voorheen.
* Aantal toestanden is nu 2057, dus vinden oplossing is triviaal.

8-koninginnenprobleem: Moraal van het verhaal Probleemformulering:

De formulering van de toestanden en acties kan een grote invloed hebben op de mogelijkheid om al dan niet een oplossing te vinden.

Routeprobleem:

* In een routeprobleem wil men een weg vinden tussen twee van 𝑛 steden. We formuleren een zoekprobleem:
* Er zijn 𝑛 toestanden, de toestand 𝑖 betekent “in stad 𝑖”.
* Beschikbare actie: RijdNaar actie.
* Wanneer men in toestand 𝑖 de RijdNaar(𝑗) actie onderneemt, dan is de volgende toestand die waarin men zich in locatie 𝑗 bevindt. Dit is het transitiemodel.
* Kost: bv. de afstand in km zijn tussen 𝑖 en 𝑗, of de gemiddelde reistijd of het gemiddeld benzineverbruik, …
* De initiële toestand is een willekeurige stad.
* De doeltest bestaat uit verifiëren dat men in de gewenste stad is aangekomen.

Rondreisprobleem:

* Elke stad minstens éénmaal bezoeken en terugkeren naar de eerste stad. Toestanden: ook bijhouden waar men geweest is!
* Toestand: huidige stad en bezochte steden: 𝑛 × 2𝑛 toestanden, dus **exponentieel** in aantal steden.
* Acties: als voorheen.
* Transitiemodel: niet enkel huidige stad aanpassen maar ook verzameling bezochte steden.
* Starttoestand: we zijn in een bepaalde stad en geen enkele stad werd reeds bezocht.
* Doeltest: terug in startstad en alle steden bezocht.
* Handelsreizigersprobleem is een speciaal geval van rondreisprobleem: elke stad moet juist éénmaal worden bezocht.

Oplossing Zoekprobleem Definitie:

* Een Oplossing Van zoek Probleem bestaat uit een **sequentie van acties** zodanig dat startend vanuit de initiële toestand een doeltoestand wordt bereikt.
* De kost van een oplossing is de som van de kosten van de individuele acties.
* Een Optimale Oplossing is een oplossing waarvoor de kost minimaal is onder alle mogelijke oplossingen.

Afbeelding met tekst, diagram, lijn, Rechthoek

Automatisch gegenereerde beschrijvingNP-complete problemen:

* Oplossen van een 𝑛2 − 1-puzzel en handelsreizigersprobleem zijn NP-complete problemen.
* Kunnen geformuleerd worden als zoekproblemen.
* We kunnen niet verwachten dat de oplossingsmethodes binnen de AI algoritmes zullen opleveren die alle instanties efficiënt kunnen oplossen!

Afbeelding met diagram, schets, ontwerp

Automatisch gegenereerde beschrijvingBoomgebaseerd Zoeken: Voorbeeld 8-puzzel:

* Initieel bevat de open lijst enkel de begintoestand.
* Kies (en verwijder) een element van de open lijst en expandeer.
  + Expanderen = genereren en toevoegen van alle opvolgers aan de open lijst.
* Open lijst bevat nu 4 elementen.
* De puzzel met de blanco bovenaan wordt gekozen voor expansie.

Boomgebaseerd Zoeken:

* Proces van kiezen uit open lijst en expanderen herhaalt zich tot
  + Een doeltoestand wordt gekozen voor expansie;
  + of de open lijst leeg is.

Boomgebaseerd Zoeken: Pseudocode

* Invoer Een zoekprobleem 𝑃.
* Uitvoer Een sequentie van acties die een oplossing is van het zoekprobleem of error wanneer er geen oplossing werd gevonden.

1: function TREESEARCH(𝑃)

2: 𝑓 ← nieuwe lege lijst # De open lijst

3: 𝑓.ADD(nieuw plan gebaseerd op initiële toestand 𝑃)

4: while 𝑓 ≠ ∅ do

5: 𝑐 ← 𝑓.CHOOSEAnDREmOVEPLAn # Kies volgend plan

6: if 𝑃.gOALTEST(𝑐.gETSTATE) = true then

7: return gETACTIOnSEq(c)

8: else

9: for (𝑠, 𝑎) ∈ 𝑐.gETSTATE.gETSuCCESSORS do

10: 𝑓.ADD(nieuw plan gebaseerd op (𝑠, 𝑎) en 𝑐)

11: return error: geen oplossing gevonden # Open lijst is leeg.

12: end function

Implementatie van een Plan:

* Een plan kan geïmplementeerd worden als een klasse met 4 velden:
* De huidige toestand.
* De laatst gekozen actie 𝑎.
* De voorganger of ouder van dit plan. Een referentie naar het plan waarvan dit plan is afgeleid door het toepassen van de huidige actie 𝑎.
* De totale kost van dit plan. Traditioneel wordt deze kost met 𝑔 genoteerd.
  + Opmerking: in principe kan 𝑔 ook berekend worden, maar omdat 𝑔 door sommige algoritmes vaak wordt gebruikt wordt deze hier opgeslagen

Criteria voor Zoekalgoritmen:

* Definitie
  + Een zoekalgoritme is Compleet wanneer het algoritme, voor elk zoekprobleem met een oplossing, effectief een oplossing vindt.
* Definitie
  + Een zoekalgoritme is Optimaal wanneer het niet enkel een oplossing vindt maar steeds een optimale oplossing teruggeeft voor elk zoekprobleem met een oplossing.

Criteria voor Zoekalgoritmen:

* Definitie
  + De Tijds Complexiteit van een zoekalgoritme bepaalt de uitvoeringstijd van het algoritme. We nemen aan dat de uitvoeringstijd evenredig is met het aantal gegenereerde toppen.
* Definitie
  + De Ruimt Complexiteit van een zoekalgoritme bepaalt de hoeveelheid geheugen die het algoritme nodig heeft tijdens de uitvoering. Dit wordt meestal uitgedrukt als het maximaal aantal toestanden dat gelijktijdig moet worden bijgehouden

Afbeelding met lijn, diagram

Automatisch gegenereerde beschrijvingGrootheden voor Ruimte- en Tijdscomplexiteit:

* De vertakkingsfactor 𝑏: Deze geeft het maximaal aantal opvolgers van een top in de zoekboom.
* 𝑑: diepte van de meest ondiepe doeltop.
* 𝑚: maximale lengte (= aantal acties) van een pad in de toestandsruimte.

Eigenschappen Zoekboom:

* Het aantal toppen in een zoekboom met vertakkingsfactor 𝑏 en maximale diepte 𝑚 wordt gegeven door (𝑏 𝑚+1 – 1) / (𝑏 – 1) = 𝑂(𝑏𝑚).
* Bovendien: de laatste laag bevat 𝑏 𝑚 toppen. Dat is méér dan alle voorgaande lagen samen!

Afbeelding met tekst, schermopname, diagram, lijn

Automatisch gegenereerde beschrijving

Graafgebaseerd Zoeken: Herhaalde Toestanden:

Boomgebaseerd zoeken onthoudt niet waar het reeds geweest is.

Graafgebaseerd onthoudt waar hij al is geweest.

Graafgebaseerd zoeken Basisidee Graafgebaseerd Zoeken:

* Expandeer elke toestand hoogstens éénmaal.
* Hoe? Houd een lijst (verzameling) bij van geëxpandeerde toestanden: de gESLOTEn LIJST.

Pseudocode:

* **Invoer** Een zoekprobleem 𝑃.
* **Uitvoer** Een sequentie van acties of error

1: function GRAPHSEARCH(𝑃)

2: 𝑓 ← nieuwe lege lijst # De open lijst

3: closed ← ∅ # Verzameling geëxpandeerde toestanden

4: 𝑓.ADD(nieuw plan gebaseerd op initiële toestand 𝑃)

5: while 𝑓 ≠ ∅ do

6: 𝑐 ← 𝑓.CHOOSEAnDREmOVEPLAn # Kies het volgende plan

7: if 𝑃.gOALTEST(𝑐.gETSTATE) = true then

8: return gETACTIOnSEq(c)

9: else

10: if 𝑐.gETSTATE ∉ closed then

11: closed ← closed ∪ 𝑐.gETSTATE

12: for (𝑠, 𝑎) ∈ 𝑐.gETSTATE.gETSuCCESSORS do

13: 𝑓.ADD(nieuw plan gebaseerd op (𝑠, 𝑎) en 𝑐)

14: return error: geen oplossing gevonden

15: end function

Blinde Zoekmethoden:

* Blinde zoekmethoden: kunnen enkel gebruikmaken van de informatie die verschaft wordt door de definitie van het zoekprobleem. De blinde zoekmethoden die we zien zijn:
  + Breedte Eerst Zoeken
  + Diepte Eerst Zoeken
  + Diepte Gelimiteerd Zoeken/Iteratief Verdiepen
  + Uniforme Kost Zoeken.
* Veel hiervan is zeer gelijkaardig aan de algoritmes die we reeds zagen in het hoofdstuk m.b.t. grafen.

Afbeelding met schets, diagram, schermopname, Rechthoek

Automatisch gegenereerde beschrijvingBreedte Eerst Zoeken:

* De open lijst is een wachtrij, i.e. een FIFO structuur.
* De zoekboom wordt systematisch laag per laag opgebouwd.
* Eerst de buren, dan pas de buren van buren

Illustratie Breedte Eerst op Binaire Boom:

De doeltoestand is 𝐷.

Eerst A, dan B, dan C, dan D, E, F, G …

Afbeelding met schets, wit, ontwerp, patroon

Automatisch gegenereerde beschrijving

Eigenschappen Breedte Eerst:

* Compleet?
  + Ja, want laag per laag.
* Optimaal?
  + Vindt steeds oplossing met minimaal aantal acties. Is niet noodzakelijk de laagste kost wanneer acties verschillende kosten hebben.
  + Er is misschien een oplossing met een dieper pad
* Tijdscomplexiteit:
  + exponentieel: 𝑂(𝑏𝑑+1).
* Ruimtecomplexiteit (hoeveel geheugen heb je nog op het slechtste moment):
  + volledige laag op diepte 𝑑 + 1: 𝑂(𝑏𝑑+1)
* Graafgebaseerd zoeken kan veel tijd winnen bij herhaalde toestanden. Breedte eerst meestal in graafgebaseerde versie.

Afbeelding met schermopname, diagram, ontwerp

Automatisch gegenereerde beschrijvingDiepte Eerst Zoeken:

Duaal aan breedte eerst: gebruikt LIFO structuur voor open lijst

Eerst A, dan B, dan de kinderen van E, dan de kinderen van N. Eenmaal N klaar is, afsluiten en overgaan naar de volgende buur.

Afbeelding met diagram, lijn, cirkel, wit

Automatisch gegenereerde beschrijving

Afbeelding met schets, wit, diagram, cirkel

Automatisch gegenereerde beschrijving

Diepte Eerst Zoeken

* Compleet?
  + Neen: kan vastraken in oneindige lus; kan oneindig lange actiesequenties nemen.
* Optimaal?
  + Neen. Vindt steeds “meest linkse” doelknoop.
* Tijdscomplexiteit:
  + exponentieel, nl. 𝑂(𝑏𝑚) in slechtste geval.
* Ruimtecomplexiteit:
  + lineair: 𝑂(𝑏 \* 𝑚).
* Diepte eerst wordt normaalgezien uitgevoerd in boomgebaseerde vorm.

Iteratief Verdiepen

* Diepte gelimiteerd zoeken: breek zoekproces diepte eerst af als vooraf bepaalde diepte 𝑑 werd bereikt.
  + Doe alsof toppen op diepte 𝑑 geen opvolgers hebben.
* Vaak weet men geen goede grens voor 𝑑.
  + Oplossing: 𝑑 systematisch verhogen. Dit is ITERATIEF VERDIEP

Iteratief Verdiepen:

* **Invoer:** Een zoekprobleem 𝑃
* **Uitvoer:** Een sequentie van acties wanneer een oplossing werd gevonden of “error” wanneer er geen oplossing werd gevonden.

1: function iterativeDeepening(𝑃)

2: 𝑑 ← 0

3: sol ← diepteGelimiteerdZoeken(𝑃, 𝑑)

4: while sol = “hit boundary” do

5: 𝑑 ← 𝑑 + 1

6: sol ← diepteGelimiteerdZoeken (𝑃, 𝑑)

7: return sol # Oplossing of error

8: end function

Diepte Gelimiteerd Zoeken:

* **Invoer** Een zoekprobleem 𝑃, een maximale diepte 𝑑.
* **Uitvoer** Een sequentie van acties wanneer een oplossing werd gevonden met diepte 𝑑 of minder; een “hit boundary” conditie wanneer tijdens het zoekproces de maximale diepte werd bereikt of “error” wanneer er geen oplossing werd gevonden.

1: function DEPTHLImITEDSEARCH(𝑃, 𝑑)

2: 𝑐 ← nieuw plan gebaseerd op initiële toestand 𝑃

3: return DLSRECuRSIVE(𝑐, 𝑃, 𝑑)

4: end function

DLSRecursief (dit moet je niet kunnen schrijven maar wel snappen):

* Invoer Een huidig plan 𝑐, een zoekprobleem 𝑃 en een maximale diepte 𝑑.
* Uitvoer Een sequentie van acties wanneer een oplossing werd gevonden met diepte 𝑑 of minder startend vanaf het huidig plan; een “hit boundary” conditie wanneer tijdens het zoekproces de maximale diepte werd bereikt of “error” wanneer er geen oplossing werd gevonden.

1: function DLSRECuRSIVE(𝑐, 𝑃, 𝑑)

2: if 𝑃.gOALTEST(𝑐.gETSTATE) = true then

3: return gETACTIOnSEq(c) # Oplossing gevonden

4: if 𝑑 = 0 then

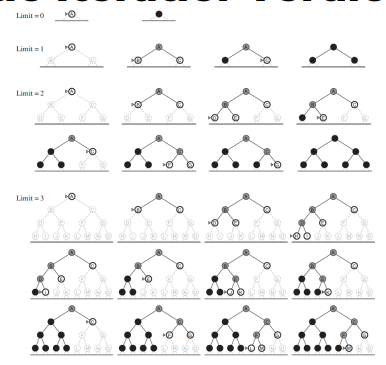
5: return “hit boundary” # Grens bereikt

6: boundaryHit ← false # Grens bereikt in één van de rec. oproepen?

7: for (𝑠, 𝑎) ∈ 𝑐.gETSTATE.gETSuCCESSORS do

8: child ← nieuw plan gebaseerd op (𝑠, 𝑎) en 𝑐

9: sol ← DLSRECuRSIVE(child, 𝑃, 𝑑 − 1) # Recursieve oproep

10: if sol = “hit boundary” then

11: boundaryHit ← true

12: else

13: if sol ≠ “error: geen oplossing gevonden” then

14: return sol # Effectieve oplossing gevonden

15: if boundaryHit = true then

16: return “hit boundary”

17: else

18: return “error: geen oplossing gevonden” 50/92

19: end function

iteratief Verdiepen: Hoeveelheid “Overbodig” Werk?

* Omdat de laatste laag het meeste aantal toppen bevat blijft de hoeveelheid werk dat “te veel” wordt gedaan binnen de perken.
* Cijfervoorbeeld: voor 𝑑 = 10 en 𝑏 = 4.
* Aantal toppen gegenereerd in laatste iteratie: (4 11 – 1) / (4 – 1) = 1 398 101.
* Aantal toppen gegenereerd in alle voorgaande iteraties 9 ∑ 𝑖=0 (4 𝑖+1 – 1) / (4 – 1) = 466 030.

Eigenschappen Iteratief Verdiepen:

* Compleet? Ja.
* Optimaal? Steeds meest ondiepe oplossing, dus niet optimaal als acties een verschillende kost kunnen hebben.
* Tijdscomplexiteit? Exponentieel 𝑂(𝑏𝑑 )
* Ruimtecomplexiteit? **lineair**!

Uniforme Kost Zoeken:

* Implementeer open lijst als prioriteitswachtrij.
  + Lagere padkost = hogere prioriteit **(binaire hoop)**.
  + Dus, plan met laagste 𝑔-waarde **(kost van de boog)** komt als volgende aan de beurt. Zelfde idee als algoritme van Dijkstra.
* Plaats doeltest in het algoritme is hier zeer belangrijk!

Afbeelding met lijn, diagram, cirkel

Automatisch gegenereerde beschrijving

Uniforme Kost (via dijkstra) Zoeken: Belang Plaats Doeltest:

Pas graafgebaseerd uniforme kost zoeken toe om van 𝑆 naar 𝐵 te gaan.

Geïnformeerde Zoekmethode:

Basisidee: Gebruik **domeinkennis** (heuristieken) om het zoeken efficiënter te laten verlopen.

Heuristieken:

* Definitie Een heuristiek ℎ is een afbeelding van de verzameling toestanden 𝑆 naar de verzameling van niet-negatieve reële getallen ℝ + , i.e. ℎ ∶ 𝑆 → ℝ+ ∶ 𝑠 ↦ ℎ(𝑠).
* Opmerking: een heuristiek heeft in de context van zoekalgoritmes dus een specifieke betekenis.
* Een goeie schatting gaat zo dicht mogelijk bij de oplossing zitten, maar is alsnog een onderschatting.

Eigenschappen van Goede Heuristiek:

* Een goede heuristiek geeft een goede indicatie voor de werkelijke optimale kost naar het doel.
* Een heuristiek moet ook snel te berekenen zijn.

Afbeelding met lijn, Rechthoek, plein, diagram

Automatisch gegenereerde beschrijvingHeuristieken voor 8-Puzzel:

* Hoeveel acties zijn deze puzzels van elkaar verwijderd?
* Je moet kijken welke getallen niet op zijn plaats staan, hier 8 vakken minimum.
* **Nog beter:** is dat je de manhemtumdistance kan gebruike (1 moet 3x verplaatsten, 2 moet 1 vak verplaatsten, 3 moet 2x verplaatsen …)
* ℎ1 = aantal niet-lege vakjes dat niet op zijn juiste plaats staat of ook ℎ2 = 8 ∑ 𝑖=1 (Manhattan afstand van vakje 𝑖 tot zijn correcte plaats).
* Wat is de waarde van ℎ1 en ℎ2 voor de zonet gegeven puzzels?
  + 21

Toelaatbare Heuristieken:

* Definitie: Een heuristiek ℎ ∶ 𝑆 → ℝ+ is **toelaatbaar** als voor elke toestand 𝑠 geldt dat ℎ(𝑠) ≤ 𝐶∗ (𝑠) waarbij 𝐶 ∗ de kost van een optimale oplossing voorstelt van 𝑠 naar een doeltoestand.
* Eigenschap: Wanneer de heuristiek ℎ toelaatbaar is, dan is ℎ(𝑔) = 0 (ik moet kleiner zijn dan de kost van h1) voor elke doeltoestand 𝑔.

Consistente Heuristieken:

* Definitie Een heuristiek ℎ ∶ 𝑆 → ℝ+ is consistent als voor elke doeltoestand 𝑔 geldt dat ℎ(𝑔) = 0 en als bovendien voor elke toestand 𝑠 en elke actie 𝑎 op 𝑠 met 𝑠 ′ = 𝑇 (𝑠, 𝑎) geldt dat
* ℎ(𝑠) ≤ 𝑐(𝑠, 𝑎, 𝑠′ ) + ℎ(𝑠′ ).
* Voorbeeld a (2) 🡪 b (5) 🡪 c
* h=10 h = 3 h = 0
* Dit kan niet want 10 <= 2 + 3 dit is niet consistent
* Mocht het 4 of 5 <= 2 + 3, dit is wel consistent
* Afbeelding met lijn, wit, diagram

  Automatisch gegenereerde beschrijvingDe schatting max maximaal zakken met de kost van de boog
* De schatting moet nauwkeuriger worden als je dichter komt bij je doel

Verband Toelaatbaarheid en Consistentie Eigenschap:

* Als een heuristiek consistent is, dan is ze ook onmiddellijk toelaatbaar.
* Omgekeerd is dit niet waar! Er bestaan toelaatbare heuristieken die niet consistent zijn.

Gulzig Beste Eerst:

* Open lijst is prioriteitswachtrij en kleinere waarde voor heuristiek ℎ betekent een grotere prioriteit.
* Afbeelding met diagram, lijn

  Automatisch gegenereerde beschrijvingKijkt enkel naar de heuristiek

Gulzig Beste Eerst is niet Optimaal:

Ga van 𝐴 naar 𝐺 m.b.v. het gulzig beste eerst algoritme.

Gulzig Beste Eerst is Niet Compleet:

* Ga van 𝐴 naar 𝐺 met **boomgebaseerd** gulzig beste eerst.
* Wat gebeurt er?
  + De schatting verbeterd alleen maar / hij pakt altijd de kleinste heuristiek

A\* -Zoeken:

* Uniforme Kost Zoeken: houdt enkel rekening met afgelegde weg 𝑔
* Gulzig Beste Eerst: houdt enkel rekening met heuristiek ℎ.
* Bij A\* -zoeken worden beide gecombineerd. Het plan met de laagste waarde voor **𝑓 = 𝑔 + ℎ** wordt als eerste geëxpandeerd.
* A\* is optimaal en compleet

Afbeelding met diagram, lijn, tekst, kaart

Automatisch gegenereerde beschrijvingA ∗ met Toelaatbare Heuristiek:

* Ga van Arad naar Bucharest.
* Gebruik de afstand in vogelvlucht tot Bucharest als heuristiek.

Afbeelding met tekst, schermopname, nummer, Lettertype

Automatisch gegenereerde beschrijving

Afstand in vogelvlucht tot Bucharest:

Opgebouwde zoekboom:

Afbeelding met tekst, schermopname, diagram, Lettertype

Automatisch gegenereerde beschrijving

Afbeelding met lijn, diagram

Automatisch gegenereerde beschrijving

A\* met een Ontoelaatbare Heuristiek:

* Ga van 𝐴 naar 𝐺 m.b.v. A\*
* Wat merk je op?
  + A\* vindt de oplossing als hij met een consistente toelaatbare heuristiek, maar niet altijd … hangt af of het boom- of graafgebasseerd is.
* F = ~~[(A, 0 + 4),~~ (A 🡪 B, 0 + 1 + 8), (A 🡪 C, 0 +3 + 1), (A 🡪 C 🡪 G, 3 + 5 + 0]

Boomgebaseerde A∗ met een Toelaatbare Heuristiek is Optimaal Eigenschap:

* Wanneer boomgebaseerde A∗ gebruikmaakt van een toelaatbare heuristiek ℎ en wanneer alle acties een kost hebben groter of gelijk aan een zekere strikt positieve 𝜖, dan is A∗ compleet en optimaal, i.e. dan vindt het algoritme steeds een optimale oplossing wanneer die bestaat.
* Eigen: graafgebasseerd vind altijd de optimale oplossing

Kern van het Bewijs:

* Er is steeds een uitbreidbaar plan.
* De 𝑓**-waarde** van zo’n plan **𝑝** is hoogstens de optimale kost vanaf de starttoestand **(f is altijd kleiner dan de kost van het pad):**
  + 𝑓(𝑝) = 𝑔(𝑝) + ℎ(𝑝) ≤ 𝑔(𝑝) + 𝐶∗ (𝑝) toelaatbaardheid ℎ = 𝐶∗ (𝑠0 ). 𝑠0 is de starttoestand
* Een plan **𝑛** met een suboptimaal pad (ander) naar een doeltoestand wordt **nooit gekozen** (vanwege de toelaatbaarheid):
  + 𝑓(𝑛) = 𝑔(𝑛) + ℎ(𝑛) = 𝑔(𝑛) 𝑛 is doeltoestand en ℎ is toelaatbaar > 𝐶∗ (𝑠0 ) want 𝑛 is suboptimaal ≥ 𝑓(𝑝) zie voorgaande berekening
* Het algoritme eindigt want aantal plannen met kost ≤ 𝐶∗ (𝑠0 ) is eindig.

A ∗ -Zoeken: Conclusie:

* Onder “milde” restricties op heuristiek is A∗ compleet en optimaal.
* Tijds- en ruimtecomplexiteit kan in slechtste geval exponentieel zijn.
  + In het algemeen sneller tekort aan geheugen dan aan tijd.
* Welke toppen worden geëxpandeerd? Alle toppen met 𝑓(𝑛) = 𝑔(𝑛) + ℎ(𝑛) < 𝐶∗ (𝑠0 ).
* Geen enkele top met 𝑓(𝑛) > 𝐶∗ (𝑠0 ) wordt geëxpandeerd. Toppen met 𝑓(𝑛) = 𝐶∗ (𝑠0 ): sommige wel, sommige niet.
* Hoe beter (groter) de heuristiek hoe minder toppen worden geëxpandeerd, maar let op ....

Ontwerpen van Heuristieken:

* Een goede heuristiek kan positieve invloed hebben op tijds- en ruimtecomplexiteit.
* We bekijken twee manieren om heuristieken te ontwerpen:
  + Gebruik van vereenvoudigde problemen.
  + Gebruik van patroondatabanken.

Vereenvoudigde Problemen: Voorbeeld Doolhof:

* Vereenvoudigd probleem voor doolhof: denk de muren weg.
* Oplossing nu onmiddellijk gekend: Manhattan-afstand tot het doel.
* Dit is een aanvaardbare (en consistente) heuristiek voor het oorspronkelijke probleem.

Vereenvoudigde Problemen: de 8-Puzzel:

* Regels 8-puzzel: 𝐴 kan naar 𝐵 worden verplaatst als
  + de vakjes 𝐴 en 𝐵 aangrenzend zijn; en
  + het vakje 𝐵 is het lege vakje.
* Laat nu één of meer van deze restricties weg:
  + 𝐴 kan naar 𝐵 worden verplaatst als 𝐴 en 𝐵 aangrenzend zijn.
  + 𝐴 kan naar 𝐵 worden verplaatst als 𝐵 het lege vakje is.
  + 𝐴 kan naar 𝐵 worden verplaatst (zonder voorwaarden).
* Welke (gekende) heuristieken krijgen we?

Afbeelding met Rechthoek, plein, lijn

Automatisch gegenereerde beschrijvingPatroon Databanken:

Bekijk een deelprobleem en sla waarde oplossing deelprobleem op in databank.

Afbeelding met schets, diagram, Plan, Technische tekening

Automatisch gegenereerde beschrijvingPatroondatabank Opbouwen?

Door achterwaarts (omkeerbare acties) te zoeken, bv. met breedte eerst:

Heuristieken Combineren:

* Heuristieken kunnen gecombineerd worden m.b.v. max-operator.
  + ℎ(𝑠) = max(ℎ1 (𝑠), ℎ2 (𝑠)).
* Dit levert een nieuwe, betere, heuristiek op.
* Aanvaardbaar als ℎ1 en ℎ2 aanvaardbaar zijn

Je moet kunnen weten wanneer je A\* gebruikt via graaf- of boomgebasseerd

# Hoofdstuk 7: Complexiteitstheorie

## Klassa P

De Complexiteitsklasse P (Polynoom wat meerderheid betekent):

* Definitie De klasse P is de verzameling van alle problemen die in polynomiale tijd kunnen opgelost worden door een (deterministisch) algoritme.
* Opmerking:
  + Polynomiale tijd betekent dat de uitvoeringstijd van het algoritme 𝑂(𝑛𝑘 ) is, met 𝑛 de grootte van de invoer en waarbij 𝑘 een constante is (onafhankelijk van 𝑛).
  + De klasse P is dus de klasse van de gemakkelijke problemen

Voorbeelden uit de Klasse P:

* Kortste pad in graaf met positieve gewichten.
* Minimale kost opspannende boom.
* Topologisch sorteren.
* Sorteren van een array van objecten …

Turing’s Stopprobleem:

* Turing’s stopprobleem:
  + Schrijf een algoritme 𝐴 (programma) dat bepaalt of een willekeurig programma 𝑃 met als input 𝐼 stopt of niet.
* Men kan bewijzen dat het programma 𝐴 niet kan geschreven worden! Turing’s stopprobleem is onbeslisbaar.
* Stelling
  + Turing’s stopprobleem is onbeslisbaar.

Schets van Bewijs (niet kennen):

* We geven een bewijs uit het ongerijmde.
* Stel dat 𝐴 wel bestaat.
  + 𝐴(𝑃, 𝐼) = { 1 als programma 𝑃 stopt voor invoer 𝐼 | 0 als programma 𝑃 niet stopt voor invoer 𝐼.
* Dan construeren we programma 𝑄
  + : 𝑄(𝑃) = { stopt als 𝐴(𝑃, 𝑃) = 0 stopt niet als 𝐴(𝑃, 𝑃) = 1.
* Merk op: als 𝐴 bestaat dan is het schrijven van 𝑄 zeer eenvoudig.

Schets van Bewijs (Vervolg, niet kennen):

* Vervolgens voeren we 𝑄 uit met zichzelf als invoer:
* 𝑄(𝑄) = { stopt als 𝐴(𝑄, 𝑄) = 0, i.e. programma 𝑄 stopt niet met zichzelf als invoer en stopt niet als 𝐴(𝑄, 𝑄) = 1, i.e. programma 𝑄 stopt wel met zichzelf als invoer.
* Beide gevallen leveren een contradictie, dus 𝐴 bestaat niet.

Voorbeeld Reducties:

* Om de mediaan te berekenen kan men de rij sorteren. Bepalen van de mediaan reduceert tot het sorteren van de rij.
* Om de kortste afstand te vinden tussen alle paren van knopen kunnen we het algoritme van Dijkstra 𝑛 keer aanroepen. Het vinden van de korste afstand tussen alle paren van knopen reduceert tot het 𝑛 keer aanroepen van het algoritme van Dijkstra.

Definitie Reductie:

* Definitie Een probleem 𝜋1 reduceert tot een probleem 𝜋2 wanneer een polynomiaal algoritme voor 𝜋2 kan gebruikt worden om probleem 𝜋1 op te lossen in polynomiale tijd.
* Dit betekent dat 𝜋2 ∈ P ⟹ 𝜋1 ∈ P
* of ook 𝜋1 ∉ P ⟹ 𝜋2 ∉ P.
* In woorden: als 𝜋1 reduceert tot 𝜋2 dan is 𝜋2 minstens zo moeilijk als 𝜋1

Compleetheid:

* Een probleem is COmPLEET voor een klasse van problemen als het tot die klasse behoort en minstens zo moeilijk is als alle problemen uit die klasse.
* (Definitie) Als 𝐶 een klasse van problemen is dan is een probleem 𝜋 𝐶-compleetheid als en slechts als 𝜋 tot de klasse 𝐶 behoort en alle problemen uit de klasse 𝐶 reduceren naar 𝜋.

## Klasse NP

De Klasse NP:

* We wensen aan te tonen dat het handelsreizigersprobleem “moeilijk” is.
* Het handelsreizigersprobleem kan opgelost worden met een exponentieel algoritme, i.e. door in essentie alle mogelijkheden te proberen.
* Informeel beschrijft de klasse **NP (Niet-Polynomiaal)** alle problemen die kunnen opgelost worden m.b.v. een exponentieel algoritme.
* De klasse wordt echter gedefinieerd in termen van efficiëntie verificatie.

De klasse NP: Definitie (zeker weten):

De klasse NP bestaat uit de problemen waarvoor oplossingen een lengte hebben die hoogstens polynomiaal is in de lengte van de invoer en waarvoor de correctheid van een oplossing kan geverifieerd worden in polynomiale tijd.

NP-Compleetheid:

* Alles wat nodig is om tot de klasse NP te behoren is dat men op een efficiënte manier oplossingen kan herkennen en dit betekent dat de klasse NP enorm veel problemen omvat.
* Wanneer een probleem NP-compleet is dan betekent dit (door definitie van compleetheid) dat dit probleem minstens zo moeilijk is als alle problemen in NP.
* Cruciale vraag: bestaan er wel NP-complete problemen?

Bestaan van NP-Complete Problemen:

* Cook (1971) en Levin (1973) toonden onafhankelijk van elkaar aan dat er NP-complete problemen bestaan.
* Karp (1972) gaf een lijst van 21 NP-complete problemen.
* Sindsdien is bewezen van honderden problemen dat ze NP-compleet zijn. Onder andere ook het handelsreizigersprobleem en het knapzakprobleem zijn NP-compleet.

NP-Compleetheid Aantonen:

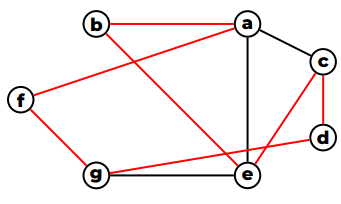
* Aantonendat een probleem 𝜋 NP-compleet is bestaat uit twee stappen.
  + 1. Aantonen dat het probleem tot de klasse NP behoort. (Dit is meestal eenvoudig om in te zien.)
  + 2. Vind een gekend NP-compleet probleem en toon aan dat dit reduceert tot het probleem 𝜋.
* Opmerking! Let op de richting van de reductie.

Probleem TSP: Formele Definitie:

* Gegeven een gewogen complete ongerichte graaf 𝐺 = (𝑉, 𝐸) en een reëel getal 𝑑, bestaat er een rondreis die alle knopen juist éénmaal bezoekt zodanig dat de kost van de rondreis hoogstens 𝑑 is.
* We wensen aan te tonen dat probleem TSP NP-compleet is.
* Opmerking: TSP staat voor Travelling Salesman Problem

TSP Behoort tot NP:

* Gegeven een voorstel van rondreis dan is het eenvoudig om
  + 1. Te controleren dat het een geldige rondreis is.
  + 2. Te verifiëren dat de kost van de rondreis hoogstens 𝑑 is.
* Dit kan uitgevoerd worden in lineaire tijd

Hamiltoniaanse Cykel Definitie:

* Een hamiltiaanse cykel in een ongerichte graaf is een cykel die elke knoop precies éénmaal bezoekt (behalve de start- en eindknoop).
* Een Hamiltoniaanse cykel (in het rood) in een ongewogen, ongerichte graaf

Probleem HC:

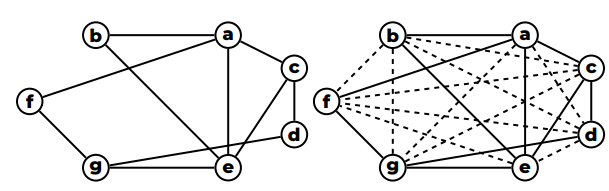
* We introduceren een probleem waarvan reeds bewezen is dat het NP-compleet is. Het staat immers in Karp’s lijst van 21 NP-complete problemen.
* Gegeven een ongewogen en ongerichte graaf 𝐺 = (𝑉, 𝐸). Bestaat er een Hamiltoniaanse cykel in 𝐺?

HC Reduceert tot TSP:

* Gegeven een graaf 𝐺 = (𝑉, 𝐸) waarop we HC willen toepassen. Construeer nu een nieuwe (gewogen) en complete graaf 𝐺 ′ = (𝑉, 𝐸′ ) (met dezelfde knopenverzameling) als volgt:
* gewicht(𝑒′ 𝑖,𝑗) = { 1 als knopen 𝑖 en 𝑗 adjacent zijn in 𝐺 | 2 als knopen 𝑖 en 𝑗 niet adjacent zijn in 𝐺.
* De constructie van graaf 𝐺 ′ kan gedaan worden in polynomiale tijd.

Voorbeeld constructie graaf G’:

De bogin in steppellijn rechts hebben gewicht 2



## Tijdscomplexiteit in volgorde

Afbeelding met tekst, schermopname, lijn, Lettertype

Automatisch gegenereerde beschrijving