Datastructuren en algoritmen
Uitwerkingen voorbeeldserie huiswerkopgaven

OPGAVE 1

Gebaseerd op opgave C-4.11 uit het tekstboek (blz. 185).

10 punten

a Ga er van uit dat polynomen worden opgeslagen in arrays, dat wil zeggen dat coëfficiënt a_i wordt opgeslagen als array-element met index i. Maak nu onderdeel a van C-4.11 en beschrijf de bedoelde methode in pseudocode en licht toe waarom deze $O(n^2)$ is.

5 punten

b Maak onderdeel b van C-4.11. U hoeft hierbij geen pseudocode te geven maar u moet uw antwoord wel toelichten.

UITWERKING OPGAVE 1

a Een algoritme in pseudocode voor de methode polynoom is:

Algoritme polynoom (A, n, x)
Input: Array A met n+1 reële getallen, reëel getal x
Output: de som van $A[i]\cdot x^i$ voor i = 0 .. n

```
resultaat ← 0
for i ← 0 to n do
    xi ← 1
    for j ← 1 to i do
        xi  ← xi * x
    {end for}
    resultaat ← resultaat + A[i] * xi
{end for}
```

Alle toekenningen en berekeningen zijn $O(1)$. De complexiteit wordt bepaald door de twee geneste lussen. De buitenste lus wordt n+1 keer doorlopen. De binnenste lus wordt 0 + 1 + … n keer doorlopen, ofwel $\frac{1}{2}\, n(n+1)$ keer. De complexiteit is dus $\Theta(n^2)$.

b Binnen elk haakjespaar zijn er 1 optelling en 1 vermenigvuldiging; verder is er aan het begin nog 1 optelling en 1 vermenigvuldiging; en er zijn n – 1 haakjesparen; in het totaal zijn er dus n optellingen en n vermenigvuldigingen. Conclusie: het aantal optellingen en vermenigvuldigingen is $\Theta(n)$.

OPGAVE 2

10 punten

a Maak opgave C-5.4 uit het tekstboek (blz. 218) met de volgende wijziging: in plaats van twee wachtrijen wordt één wachtrij gebruikt. Ga uit van een array–implementatie en beantwoord de opgave zoals opgave 5.2 uit het werkboek.

5 punten

b Is er bij onderdeel a sprake van een toepassing van het adapter–patroon? Licht toe waarom (niet)?

UITWERKING OPGAVE 2

a Voor de operatie pop wordt het eerste element uit de wachtrij gehaald; dit gebeurt door de operatie dequeue. De complexiteit is $O(1)$.
Voor de operatie push wordt het element voorin de wachtrij gezet; dit gebeurt door de operatie enqueue. Vervolgens worden de reeds in de wachtrij aanwezige elementen voor het nieuwe element gezet. Dit gebeurt door size() – 1 keer de operaties dequeue en enqueue aan te

roepen, waarbij het door dequeue verkregen element weer met enqueue in de wachtrij gezet wordt. Aangezien enqueue en dequeue $O(1)$ zijn, is de complexiteit van push dus $\Theta(n)$.
Voor de operatie top wordt het eerste element uit de wachtrij gehaald; dit gebeurt door de operatie front. De complexiteit is $O(1)$.
Voor size en isEmpty kunnen de gelijknamige operaties uit de queue ADT gebruikt worden, met complexiteit $O(1)$.
Opmerking:
Het kan ook anders maar minder voordelig: push met enqueue en pop en top beide met een herhaling van dequeue en enqueue op bovenbeschreven manier.

b Bij het adapter–patroon gaat het om een eenvoudige aanpassing. Voor pop en top geldt dat wel maar niet voor push (of andersom). Dus geen adapter–patroon.

OPGAVE 3

10 punten

Een array list zullen we meestal implementeren met een array. In bepaalde gevallen is een andere implementatie handiger. Een array list V heet ijl (Eng: sparse) als de meeste elementen van de array list leeg zijn, dwz nul of null, afhankelijk van het soort elementen dat de array list bevat. Als van de n elementen uit de array list er m niet leeg zijn, kunnen we de array list efficiënt opslaan door alleen de m niet-lege elementen op te slaan, samen met het rangnummer van het element in de array list. We gebruiken hiervoor een dubbelgeschakelde lijst. Als de array list het element e met index r bevat, dan slaan we in de lijst het paar (r, e) op. Beschrijf hoe bij deze implementatie de methoden van de Array list ADT uitgevoerd kunnen worden, en bepaal de complexiteit van de methodes bij deze implementatie.

UITWERKING OPGAVE 3

size() bij het bepalen van de grootte van de array list moeten we ook de null-elementen meetellen. We hebben dus een nieuw attribuut sizeArrayList nodig. Opvragen van de waarde hiervan kost O(1).

isEmpty() De array list is leeg als sizeArrayList = 0; het bepalen hiervan kost O(1)

Bij de volgende implementaties is er voor gekozen om de elementen op volgorde van index in de lijst te plaatsen; het kan ook zonder, maar dat maakt het zoeken minder efficiënt.

get (k)
Doorloop de lijst tot er een schakel (k', e) gevonden is waarvoor k' ≥ k. Als k' = k, dan is get(k) gelijk aan e, anders gelijk aan null. In het slechtste geval moet de hele lijst doorlopen worden, dit kost O(m).
set(k, e)
Doorloop de lijst tot er een schakel (k', e') gevonden is waarvoor k' ≥ k. Als k' = k en e ≠ null, vervang dan de schakel door (k, e). Als k' = k en e = null, verwijder dan de schakel (k', e'). Als k' > k en e ≠ null, voeg dan (k, e) in voor (k', e'). Als k' > k en e = nul doe dan niets.
Dit kost O(m)
add (k, e)

Doorloop de lijst vanaf het einde tot er een schakel (k', e') gevonden is waarvoor k' ≤ k en verhoog alle indices tot aan k' met 1. Hoog sizeArrayList 1 op. Als k' = k en e ≠ null, voeg dan de schakel (k, e) in vóór (k', e') en vervang k' door k' + 1. Als k' = k en e = null vervang dan k' door k' + 1. Als k' < k en e ≠ null, voeg dan (k, e) in na (k', e'). Als k' < k en e = null doe dan verder niets.
Dit kost Θ(m).
remove(k)
Doorloop de lijst vanaf het einde tot er een schakel (k', e') gevonden is waarvoor k' ≤ k en verlaag alle indices tot aan k' met 1. Verlaag sizeArrayList met 1. Als k' = k verwijder dan de schakel (k, e). Als k' < k doe dan verder niets.
Dit kost Θ(m).

OPGAVE 4

Een min-max heap is een datastructuur waarin het mogelijk is om zowel de operatie deleteMin als de operatie deleteMax in O(log *n*) uit te voeren. De structuur maakt net zo als een gewone heap gebruik van een volledige binaire boom. De knooppunten van de boom voldoen aan de volgende twee min-max heap eigenschappen:
- als de diepte van het knooppunt *k* even is, dan zijn alle sleutels van knooppunten uit de subboom met wortel *k* groter dan of gelijk aan de sleutel van dit knooppunt; de sleutel van k is dus minimaal in de bijbehorende subboom.
- als de diepte van het knooppunt *k* oneven is, dan zijn alle sleutels van knooppunten uit de subboom met wortel *k* kleiner dan of gelijk aan de sleutel van dit knooppunt; de sleutel van *k* is dus maximaal in de bijbehorende subboom.
(Voor de volledigheid vermelden we dat de wortel van de heap een even diepte heeft , namelijk 0.)

5 punten

a Teken een min-max heap voor de sleutels 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

7 punten

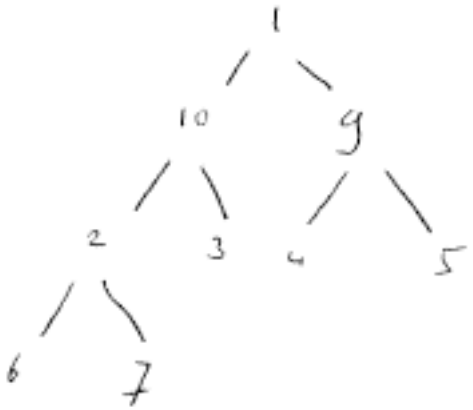
b Geef een algoritme om een nieuwe knoop aan een min-max heap toe te voegen.

8 punten

c Geef algoritmes voor deleteMin en deleteMax

UITWERKING OPGAVE 4

a



b

Algoritme insert(H, v)
Input een min-max heap H en een element v
Output geen, na afloop is v opgenomen in H, zodanig dat H weer een min-max heap is.
insert v in de eerste externe knoop van H
k ← key(v)
d ← diepte van k
klaar ← false
zolang ! klaar
 klaar ← true
 als diepte is oneven
  als k < key(v.parent)
   wissel v en v.parent
   klaar = false
  als k > key(v.parent.parent)
   wissel v en v.parent.parent
   klaar = false
  d ← diepte van k
als diepte is even
  als k > key(v.parent)
   wissel v en v.parent
   klaar = false
  als k < key(v.parent.parent)
   wissel v en v.parent.parent
   klaar = false
  d ← diepte van k

c

We maken gebruik van een hulpalgoritme downbubble:

Algoritme downbubble(H, v)
Input een min-max heap H en een knoop v van H
Output geen, vanaf v wordt naar beneden toe de min-max heap eigenschap hersteld.

k ← key(v)
d ← diepte van v
klaar ← false
als beide kinderen van v external zijn
klaar ← true

```
zolang ! klaar
  klaar  ← true
  als diepte is oneven
      u ← kind van v met grootste sleutel
     als k < key(u)
        wissel v en u
         klaar ← false
    als alle kleinkinderen van  v external zijn
        klaar ← true
    u ← kleinkind van v met kleinste sleutel
    als k > key(u)
        wissel v en u
         klaar ← false
als diepte is even
      u ← kind van v met kleinste sleutel
     als k > key(u)
        wissel v en u
         klaar ← false
    als alle kleinkinderen van  v external zijn
        klaar ← true
    u ← kleinkind van v met grootste sleutel
    als k < key(u)
        wissel v en u
         klaar ← false
```

Algoritme deleteMin(H)
Input een min-max heap H
Output minimaal element, na afloop is een minimaal element verwijderd uit H, zodanig dat H weer een min-max heap is.

```
min ← element uit de wortel van H
vervang het element uit de wortel van H door het laatste element uit H
en verwijder dit laatste element
downbubble(H, wortel H)
return min
```

Algoritme deleteMax(H)
Input een min-max heap H
Output maximaal element, na afloop is een maximaal element verwijderd uit H, zodanig dat H weer een min-max heap is.

```
if beide kinderen van de wortel external zijn
 max ← element uit de nieuwe wortel van H
 vervang het element uit de wortel van H door het laatste element uit H.
else
  v ← kind van wortel met grootste sleutel
  max ← element uit v
 vervang het element uit v door het laatste element uit H en verwijder dit
  laatste element
  downbubble(H, v)
return max
```

10 punten

OPGAVE 5

Beschouw 2 verzamelingen met gehele getallen, S = $\{s_1, s_2, \ldots, s_n\}$ en T = $\{t_1, t_2, \ldots, t_m\}$.
Beschrijf twee verschillende manieren om te testen of S een deelverzameling is van T. U mag alle datastructuren uit het boek hierbij gebruiken. Het is niet de bedoeling dat u hier algoritmes in pseudocode of java code geeft, maar een beschrijving van de algoritmen in woorden. Wat is de complexiteit van elk algoritme?

UITWERKING OPGAVE 5
Er zijn vele uitwerkingen mogelijk. Hier onder volgen er enkele:

manier 1
Laat H een lege hash-tabel zijn.
Vul deze met de elementen van T (sleutels en elementen identiek) met een geschikte hash functie en bijvoorbeeld lineair probing om collisios op te lossen. De methode insertItem wordt dus m keer aangeroepen.
Controleer vervolgens met aanroepen van findElement(s) of de elementen van S in de hash-tabel voorkomen. In het slechtste geval, als S inderdaad een deelverzameling van T is, wordt de methode findElement n keer aangeroepen.
Complexiteit O(n + m).

manier 2
Sla de elementen van T op in een AVL boom. Ga voor elk element uit S vervolgens na of deze zich in de AVL boom bevindt. Als we één element uit S vinden dat zich niet in de AVL boom bevindt, dan is S geen deelverzameling van T. Als we alle elementen uit S kunnen terugvinden in de AVL boom, dan is S een deelverzameling van T.
Complexiteit:
Het opbouwen van de AVL boom heeft complexiteit O(m log m).
Het vinden van een element uit S in de AVL boom heeft complexiteit O(log m).
Het nagaan of alle elementen uit S zich in de AVL boom bevinden heeft complexiteit O(n log m).
Als n > m is, dan kan S geen deelverzameling van T zijn. In dit geval kunnen we meteen concluderen dat S geen deelverzameling van T is, en is de complexiteit $\Theta(1)$.
Als $n \leq m$, dan kan S een deelverzameling van T zijn. In dit geval moeten we het hierboven beschreven algoritme uitvoeren, waarvan de complexiteit O((m + n) log m) is. Aangezien $n \leq m$, kunnen we dit vereenvoudigen tot O(m log m).
Oftewel de complexiteit van het algoritme is O(m log m).

Manier 3:
Sla de elementen van T op in een array list en sorteer vervolgens deze array list m.b.v. quick sort. Gebruik vervolgens het binary search algoritme om elk element uit S in deze array list terug te vinden. Als we één element uit S vinden dat zich niet in de array list bevindt, dan is S geen deelverzameling van T. Als we alle elementen uit S kunnen terugvinden in de array list, dan is S een deelverzameling van T.
Complexiteit:

Het opslaan van de elementen uit T in een array list en het vervolgens sorteren van deze array list heeft complexiteit O(m log m).
Het vinden van een element uit S in de array list m.b.v. het binary search algoritme heeft complexiteit O(log m).
Het nagaan of alle elementen uit S zich in de array list bevinden heeft complexiteit O(n log m).
Als $n > m$ is, dan kan S geen deelverzameling van T zijn. In dit geval kunnen we meteen concluderen dat S geen deelverzameling van T is, en is de complexiteit $\Theta(1)$.
Als $n \leq m$, dan kan S een deelverzameling van T zijn. In dit geval moeten we het hierboven beschreven algoritme uitvoeren, waarvan de complexiteit $O((m + n) \log m)$ is. Aangezien $n \leq m$, kunnen we dit vereenvoudigen tot $O(m \log m)$.
Oftewel de complexiteit van het algoritme is $O(m \log m)$.

manier 4
We merken vooreerst op dat S en T gehele getallen bevatten. Een manier om te testen of S een deelverzameling is van T, is door eerst alle elementen van T in een array list te plaatsen, waarbij elk element k op index k in de array list wordt geplaatst. Dit is analoog aan de eerste fase van een bucket-sort. In dit geval hoeven we geen buckets te gebruiken, eventuele dubbelen in de verzameling hoeven slechts 1 keer in de array list geplaatst te worden.
Nadat de elementen van T in een array list zitten, overlopen we alle elementen s_i van S en kijken telkens of elk element s_i in de array list voorkomt. Het overlopen eindigt als één element s_i niet voorkomt in de array list.
De tijdscomplexiteit van dit algoritme is zeer goed: $O(n_t+n_s)$ en $\Omega(n_t)$, met n_t en n_s het aantal elementen van T en S respectievelijk.
De ruimtecomplexiteit kan echter uit de hand lopen als het grootste getal (N_t) van T zeer groot is in vergelijking met n_t. De array list heeft ruimtecomplexiteit $\Theta(N_t)$.

In het geval T wel zeer grote getallen bevat en S niet, kan men omgekeerd te werk gaan: de elementen van S in een array list steken en voor alle elementen van T controleren of het element in de array list voorkomt. We plaatsen een vlag in de array list op die plaats als het element van T in de array list voorkomt. Nadien overlopen we alle elementen in de array list en controleren of alle vlaggen voor alle elementen geplaatst zijn. Zo ja, dan is S een deelverzameling van T.
Tijdscomplexiteit: $O(n_s+ n_t+N_s)$ met N_s het grootste getal in S.
Ruimtecomplexiteit $\Theta(N_s)$.

manier 5
Een vijfde methode is om de elementen van T in een geordende dictionary (geïmplementeerd d.m.v. een zoekboom) te steken. Voor elk element van S controleren we vervolgens of dit element aanwezig is in de boom.
Tijdscomplexiteit: $O(n_t.\log(n_t)) + O(n_s).O(\log(n_t))$ of $O((n_t+n_s).\log(n_t))$
De ruimtecomplexiteit is hier $\Theta(n_t)$ wat zeer goed is.

Ook hier kan men omgekeerd tewerk gaan als n_t veel groter is dan n_s, op een analoge manier als in 1.
Tijdscomplexiteit: $O((n_t+n_s).\log(n_s))$
Ruimtecomplexiteit: $\Theta(n_s)$

Conclusie: factoren zoals de beperking van ruimte- en/of tijds-complexiteit in een toepassing, de grootte van de sets en de grootte van

de getallen in de sets, zullen de keuze voor de een of de andere implementatie in een bepaalde toepassing bepalen. De voorgestelde algoritmen hebben elk hun eigen voor- en nadelen.

OPMERKING: Als S een deelverzameling is van T dan geldt ns <= nt; de complexiteit kan dan alleen in nt uitgedrukt worden.

OPGAVE 6

5 punten

a Lees opgave C-13.6 uit het tekstboek.
Teken nu een boom met 8 knooppunten waarbij het centrum excentriciteit 3 heeft.

10 punten

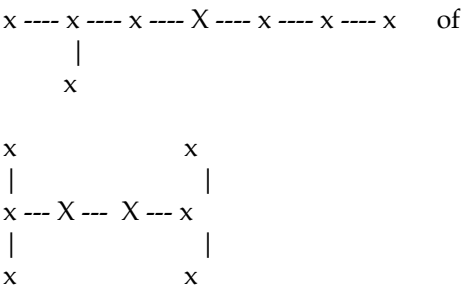
b Maak opgave C-13.6 uit het tekstboek

5 punten

c Wat is de complexiteit van uw algoritme?

Uitwerking opgave 6

a



Twee bomen waarvan het centrum (aangegeven met X) excentriciteit 3 heeft.

b Een mogelijk uitwerking gaat uit van een BFS – achtig algoritme, dat de lengte van de langste tak vanuit het startpunt s teruggeeft. Dit algoritme wordt voor elk knooppunt aangeroepen waarbij het minimum van de langste takken en het bijbehorende knooppunt wordt bijgehouden.
Dit algoritme is echter niet efficiënt: Het BFS algoritme kost O(n + m) = O(n). (in een boom geldt m = n-1).
Dit algoritme wordt n keer aangeroepen, dus de complexiteit is O(n^2).

Het volgende algoritme is efficiënt, en is gebaseerd op het volgende idee:
Verwijder de bladeren van T, resultaat is T1
Verwijder de bladeren van T1, resultaat is T2
Verwijder de bladeren van T2, resultaat is T3
Kortom
Verwijder de bladeren van Tk, resultaat is Tk+1
Stop als het resultaat 1 of 2 knooppunten heeft.
Als Tk 1 knooppunt heeft, is dit is het Centrum met excentriciteit k
Als Tk 2 knooppunten heeft, zijn er 2 centra met excentriciteit k+1.

Het algoritme in pseudocode luidt als volgt

Algorithm findCenter(T)
Input: Een tree T
Output: Een queue met de knooppunten die center zijn

```
    Laat counter een integer zijn, initieel gelijk aan T.size()
    Laat Q een initieel lege queue zijn
    for each vertrex u of T do
        Laat degree(u) de degree zijn van vertex u
        if degree(u) = 1 then
        Q.enqueue(u)
        counter = counter – 1
    while counter > 0 do
        u ← Q.dequeue()
        for each edge (u,w) of u do
            degree(w) = degree(w) – 1
            if degree(w) = 1 then
                Q.enqueue(w)
                counter = counter – 1
return Q
```

Dit algoritme is gebaseerd op het volgende:
-als de tree T meer dan twee punten bezit is een uiteinde-knooppunt nooit een centrum
- een langste pad van een knooppunt eindigt altijd in een uiteinde
Hieruit volgt dat als we van een tree T de uiteinden verwijderen het centrum nog steeds hetzelfde blijft, het langste pad is alleen 1 minder geworden in lengte. Door herhaaldelijk de uiteindes te verwijderen blijven tenslotte de centrums over.

In de begin wordt in een for-each loop de degree van ieder punt geïnitialiseerd, we slaan dat op in degree(u). Een uiteinde heeft dan precies een degree van 1, deze worden dan is de queue Q gestopt en de counter wordt één verlaagd. Daarna vinden in een while-loop de iteraties plaats. Een punt wordt uit de queue gehaald en als het ware verwijderd in de tree T door van alle omliggende aangesloten punten de degree met één te verlagen. Als de degree dan 1 geworden is dan is dit een uiteinde geworden en wordt het in de queue gestopt en de counter wordt met één verlaagd. Als uiteindelijk alle punten in de queue zijn gestopt dan is de counter gelijk aan nul. Alle punten die nu nog in de queue Q zitten zijn dan de centrums.

Is the center unique? If not, how many distinct centers can a tree have?
Het centrum hoeft niet uniek te zijn, er kunnen maximaal twee centrums bestaan. Dit is bijvoorbeeld het geval in een tree met twee knooppunten en één edge. Als er drie centrums zouden bestaan in tree T dan is er altijd wel een koppel centrums te vinden waarvoor een ander punt in het centrum van beide ligt waardoor er een tegenspraak optreedt. Het maximum is dus twee.

c De complexiteit is gelijk aan $\Theta(n)$, het lijkt namelijk heel veel op het algoritme topologicalSort(). Het initialiseren van de degrees(u) kan gedaan worden door een traversel door de tree, dit duurt $\Theta(n + m)$. Omdat m echter gelijk is aan $n - 1$ kunnen we zeggen dat dit gelijk is aan $\Theta(n)$.

Een vertex u kan in de queue worden gestopt als zijn degree gelijk is aan 1, tijdens de initialisatie of na het verminderen van de degree in een iteratie. Daarnaast gaat het algoritme door totdat iedere vertex in de queue is gestopt, totdat counter gelijk is aan 0. Ieder knooppunt komt dus precies éénmaal aan de beurt. Als een vertex uit de queue wordt gehaald worden alle edges die ermee verbonden zijn bekeken, dit

betekent dat ieder edge tweemaal aan de beurt komt (eenmaal van links en eenmaal van rechts). De degree van veel punten wordt dan op een gegeven moment op nul gezet maar dat heeft verder geen impact. Omdat iedere edge tweemaal aan de beurt komt is de complexiteit van alle iteraties gelijk aan $\Theta(2m)$, oftewel $\Theta(n)$. De totale complexiteit van de initialisatie en iteraties is hierdoor $\Theta(n)$.

10 punten

OPGAVE 7
Maak opgave C12-12 van het tekstboek.

UITWERKING OPGAVE 7
Laat S de omkering van T zijn. Pas nu KMP aan voor patroon T en string S:

```
Algoritme KMPomkering(T)
Input de string T
Output de langste prefix van T die substring is van de omkering van T
S ← omkering van T
f ← KMPfailurefunctie(T)
i ← 0
j ← 0
max ← 0
while i < n
    if T[j] = S[i]
        i ← i + 1
        j ← j + 1
        max ← maximum(max, j – 1 )
    else
        if  j > 0
            j ← f(j  – 1)
        else
            i ← i + 1
return T[0,...max]
```

KMP heeft complexiteit O(n + m), dus in dit geval (n = m) O(n). Ook het omkeren kost O(n), dus de totale kosten zijn O(n).

Let op dat in uw antwoord de motivatie voor de gevonden complexiteit niet ontbreekt.