Inhoudstafel

Reeks 1

1) Geef de twee manieren om een heap te initialiseren met hun performantie. Performantie moet je uitleggen. Niet enkel de formules.

Er zijn twee mogelijkheden.

1) *Elementen één per één toevoegen*

   Dit werkt voor zowel een ledige heaptabel als een reeds (half)opgevulde heaptabel. Aangezien een heap voorgesteld wordt als een vector, voeg je het nieuwe element achteraan toe. Bij een maxheap kijk je of de ouder kleiner is als het nieuwe element (de ouder kan gevonden worden door (i - 1)/2 met *i* = index van het nieuwe toegevoegde element. Indien de ouder kleiner is als het nieuwe element swap je de twee elementen, en kijk je opnieuw of de ouder kleiner is enz… Bij een minheap kijk je of de ouder groter is.

   Er worden *n* elementen toegevoegd, en elk element kan hoogsten helemaal tot boven gaan (swappen tot boven) ($h = log_2 n$). De performantie is $O(nlog_2 n)$

2) *Door dealheaps samen te voegen*

   Deze methode werkt alleen als er al een ingevulde tabel is. De methode begint op het voorlaatste niveau aangezien de kinderen van deze knopen een geldige dealheap zijn. Dit is logisch aangezien er maar één knoop is. Algemeen wordt gesteld dat de wortel van elke deelheap naar beneden wordt geswapt met een pad van kinderen totdat de heapvoorwaarde voldaan is. Meer specifiek als alle deelheaps op hoogte h al geheapified zijn (laagste niveau h = 0), kunnen de bomen op hoogte h +1 ge-heapified worden door de wortel naar beneden te swappen met het pad van de grootste kinderen bij een maxheap, of het pad van de laagste kinderen bij een minheap. Dit proces heeft dus maximum $O(log_2 n)$ swaps per knoop.

   De knopen op het voorlaatste niveau kunnen hoogstens 1 maal geswapt worden, op het tweede laagste niveau hoogstens 2 maal enz. Veronderstel dat het onderste niveau volledig opgevuld is krijgen we:

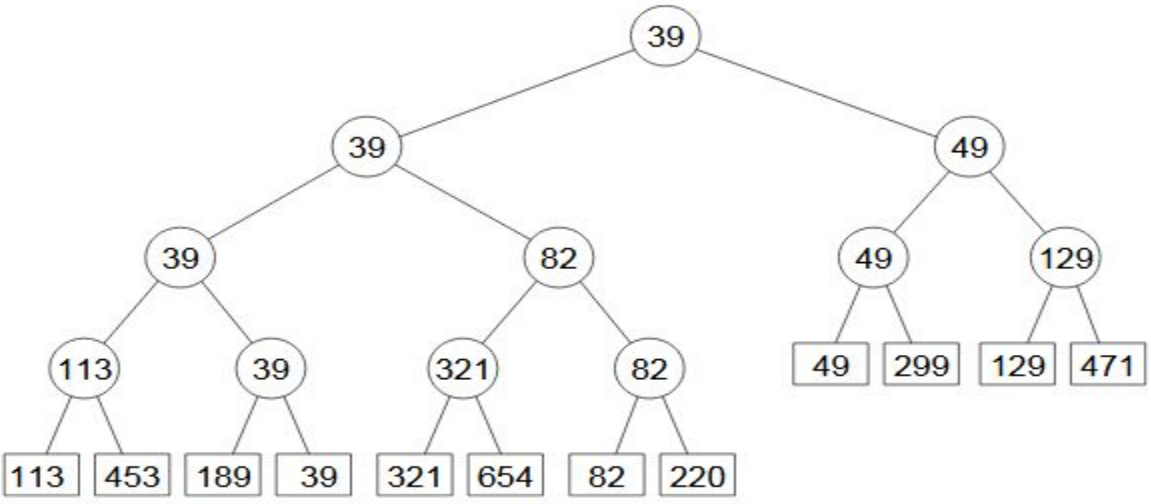
   $$T(n) = 2^{h-1} . 1 + 2^{h-2} . 2 + ... + 4(h-2) + 2(h-1) + 1 . h$$

   Dit is een afschatting van een som waarbij $T(n) \leq 2 . 2^{h-1}$. De performantie is dus $O(n)$.

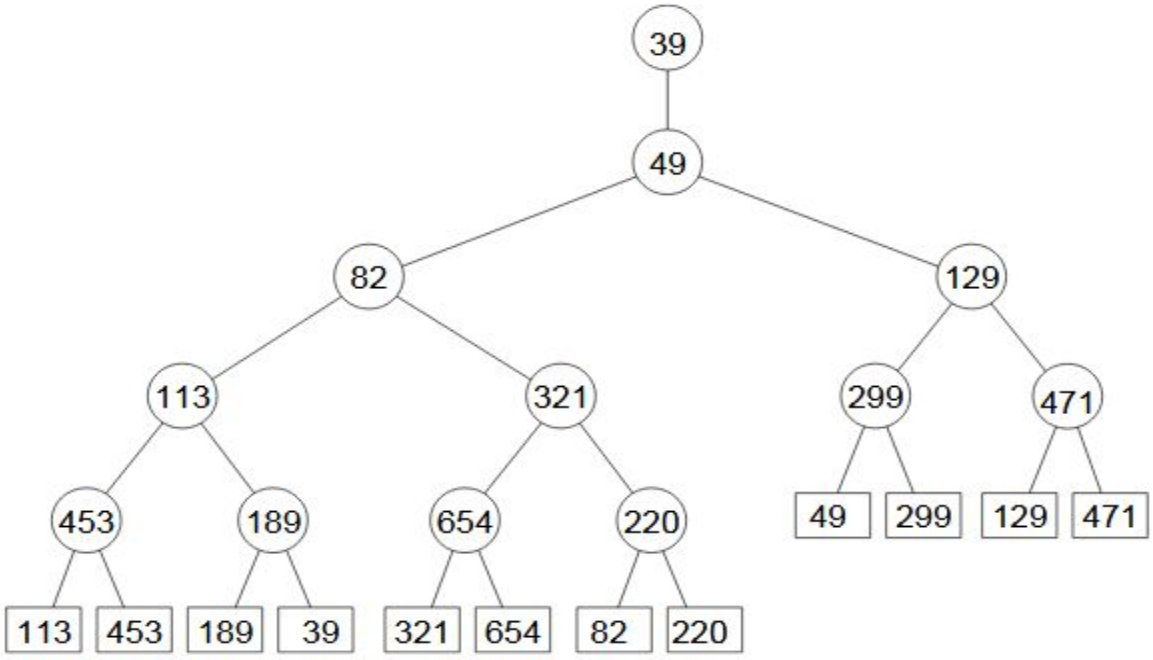
2) Leg uit wat een selectieboom is. performantie van toevoegen, verwijderen enz.. En een implementatievoorbeeld.

Een selectieboom is een complete, volle binaire boom (volle = elke knoop heeft twee kinderen, compleet = elk niveau is volledig opgevuld) waarbij de uitwendige knopen een andere functie hebben dan de inwendige. Alle gegevens zitten in de bladeren en elke inwendige knoop bevat de kleinste sleutel van zijn deelboom. Een selectieboom kan twee vormen aannemen.

1. *Selectieboom van winnaars:* In dit geval wordt elk paar kinderen met elkaar vergeleken en de kleinste sleutel zal naar hun respectievelijke ouder gekopieerd worden. Daarna wordt er een niveau hoger weer elk paar kinderen vergeleken totdat het kleinste element in de wortel zit.



2. *Selectieboom van verliezers*. In dit geval komt het grootste van de twee elementen in de ouder, maar zij kunnen niet meer deelnemen aan de volgende “wedstrijd”.



Het aanpassen van de boom gebeurd in beide gevallen langs boven en is $\theta(log_2 n)$

3)Het probleem van Clustering uitleggen. Welk algo wordt hiervoor gebruikt en de performantie ervan.

Probleem = zorgen dat objecten in dezelfde cluster dicht bij elkaar liggen en deze in verschillende clusters ver van elkaar.

Clustering kan beschouwd worden al het construeren van een ongerichte graaf, met objecten als knopen. => constructie van een MOB mbhv het algoritmen van Kruskal, in een complete graaf met objecten als knopen en afstanden als knopen.

Reeks 2

1) Geef de efficientië van heapsort en verklaar ze ook.
bijvraag: is heapsort in het beste geval veel verschillend van het slechtste geval?

Heapsort maakt gebruik van een heap. Dus vooraleer heapsort kan starten moet er een heap aangemaakt worden. De efficiëntste heapconstructie is $O(n)$ (zie reeks 1 vraag 1). De constructie moet slechts één maal uitgevoerd worden.

Meer specifiek, heap sort maakt gebruik van een maxheap, dus het grootste element zal in de wortel zitten. De waarde van de wortel halen is $O(1)$, het swappen van de wortel met het laatste element is ook $O(1)$. De heapvoorwaarde is nu wel verbroken, de nieuwe wortel moet hoogstens *h* (= hoogte van de heap) swaps uitvoeren om terug een correcte heap te vormen. Aangezien $h = log_2(n)$ is de totale uitvoeringstijd $O(nlog_2 n)$

Heap sort is zowel in het beste als het slechtste geval $O(nlog_2 n)$ aangezien de heapconstructie onafhankelijk is van de oorspronkelijke volgorde van de tabel. Verder wordt ook altijd de wortel geswapt met het laatste element in de tabel wat meestal een klein element is, kleine elementen zullen vaak diep dalen doorheen de heap wat dus ook $O(log_2 n)$ operaties oplevert.

2) Leg uit hoe we de kortste weg vanuit een knoop kunnen berekenen in een 'dag' (gerichte lusloze graaf). Leg ook de verschillende mogelijkheden van projectplanning uit bij dag's.
bijvraag: waar moeten we starten bij topologisch rangschikken en waarom bekijken we de knopen links ervan niet meer?

Rangschik eerst de graad in topologische volgorde (via diepte eerst of ingraden). Aangezien een toplogisch gesorteerde graaf alleen naar rechts wijst, kan een knoop nooit een afstand aanpassen dat links van hem ligt. Begin dus van de startknoop, overloop zijn buren in toplogische volgorde, en pas de kost aan als de kost van de reeds afgelegde weg + de kost naar de nieuwe buur. Het toplogisch rangschikken is $\Theta(n + m)$, initialisatie van voorlopers en opvolgers is $\Theta(n)$ en de buren van elke knoop komen één maal aan bod voor een bijkomende term $\Theta(n + m)$. De performantie is $\Theta(n + m)$.

Projectplanning kent 4 toepassingen:

1. *<u>Minimale projectduur:</u>* De minimale tijd van een project bepalen komt erop neer dat voor elke taak de langst mogelijke (op voorhand bepaalde) tijd doet om die taak af te werken. Het langste pad vinden in een '*dag'* kan bekomen worden door alle gewichten om te draaien van teken (vermenigvuldigen met -1) en daarop het algoritme van het korste pad zoeken toe te passen. De performantie is hier zoals kortste weg zoeken ook $\Theta(n + m)$

2. *Vroegste aanvangstijden:* De vroegste aanvangstijd is de minimale tijd nodig vooraleer een taak kan gestart worden (dus vooraleer de vorige taken die naar deze taak wijzen klaar zijn). Dit wordt berekend op een analoge manier als bij minimale projectduur.
3. *Uiterste aanvangstijden:* Dit is de tijd waarop de taak zeker moet gestart worden. Bij de eindknoop is deze gelijk aan de vroegste aanvangstijd. Voor elke andere knoop kan dit berekend worden door is dit de vroegste van de uiterste aanvangstijden van al zijn buren min de tijd van de knoop zelf.
4. *Vertraging en kritieke weg:* Met de vorige gegevens kan de vertraging die elke taak afzonderlijk mag oplopen zonder invloed te hebben op heel het project berekend worden. Dit is gelijk aan het verschil van de uiterste aanvangstijd min de vroegste aanvangstijd van een taak.

3) Wat zijn threaded trees? Welke operaties kan men er efficiënt mee uitvoeren? Bijvraag: Is het niet onefficïent dat je al die pointers naar de juiste plaats moet laten wijzen?

Een threaded tree is een speciale vorm van een binaire zoekboom waarbij ook bladeren naar een knoop verwijzen. Meer specifiek:

1. Bij knopen die twee kinderen wijst de linkerpointer naar het linkerkind en de rechterpointer naar het rechterkind.
2. Bij knopen die geen linkerkind hebben, verwijst de linkerpointer naar de voorloper van die knoop. Bij knopen die geen rechterkind hebben, verwijst de rechterpointer naar de opvolger van die knoop.
3. Bij knopen die geen kinderen hebben verwijst de linkerpointer naar de voorloper en de rechterpointer naar de opvolger van die knoop.

Een toepassing van threaded trees is het inorder overlopen van een boom. Aangezien dat knopen een verwijzing hebben naar hun opvolger is er geen recursie meer nodig en is er dus geen stapel meer nodig (= minder geheugen en sneller).

Het laten wijzen van die pointers is juist eenvoudig en heeft zelfs een performantie van $O(1)$. Stel dat we een nieuw element moet toegevoegd worden aan de linkerpointer van een knoop. De opvolger van de nieuwe knoop is duidelijk de ouder en die hebben we aangezien we die nodig hebben om de linkerpointer op te vullen. De voorloper van de nieuwe knoop wordt de voorloper van de ouder. Indien het nieuwe element een rechterkind is, wordt de voorloper de ouder en de opvolger wordt de opvolger van de ouder. Enkel in het uitzonderlijk geval dat een knoop ofwel het minimum (dus geen voorloper) ofwel het maximum(dus geen opvolger) is, verwijzen respectievelijk de voorloper en de opvolger naar een nullpointer.

Reeks 3

1. Leg mergesort samen met zijn efficiëntie uit.
Merge Sort is een typisch voorbeeld van een verdeel-en heers methode. Merge Sort splitst de originele tabel op in twee gelijke delen en sorteert deze twee onafhankelijke delen met dezelfde methode (dus elke deeltabel wordt ook opgesplitst in twee totdat elke deeltabel grootte 1 is, want die zijn gesorteerd). Nadien moeten de tabellen terug samengevoegd worden. Dit is eenvoudig aangezien beide tabellen gerangschikt zullen zijn. Je vergelijkt telkens de kleinste elementen van beide deeltabellen en neemt het minimum weg. Indien het einde van een deeltabel bereikt wordt kan de rest van de andere deeltabel gekopieerd worden.

De performantie is het gemakkelijkst te onderzoeken indien *n* een macht van twee is, want dan wordt elke deeltabel perfect in twee gesplitst. Er kan dus gezegd worden dat

$$T(n) = 2T(n/2) + cn$$

Waarbij *cn* de tijd is om de tabellen terug samen te voegen. Samenvoegen van twee gerangschikte tabellen is $\Theta(n_1 + n_2)$. Als je beide leden deelt door *n* krijg je de algemene vorm van een logaritmische afschatting:

$$\frac{T(n)}{n} = \frac{2T(n/2)}{n} + c$$

Met $f(n) = \frac{T(n)}{n}$. Op pagina 13 formule 2.5 is duidelijk te zien dat dit $O(log_2 n)$ is.

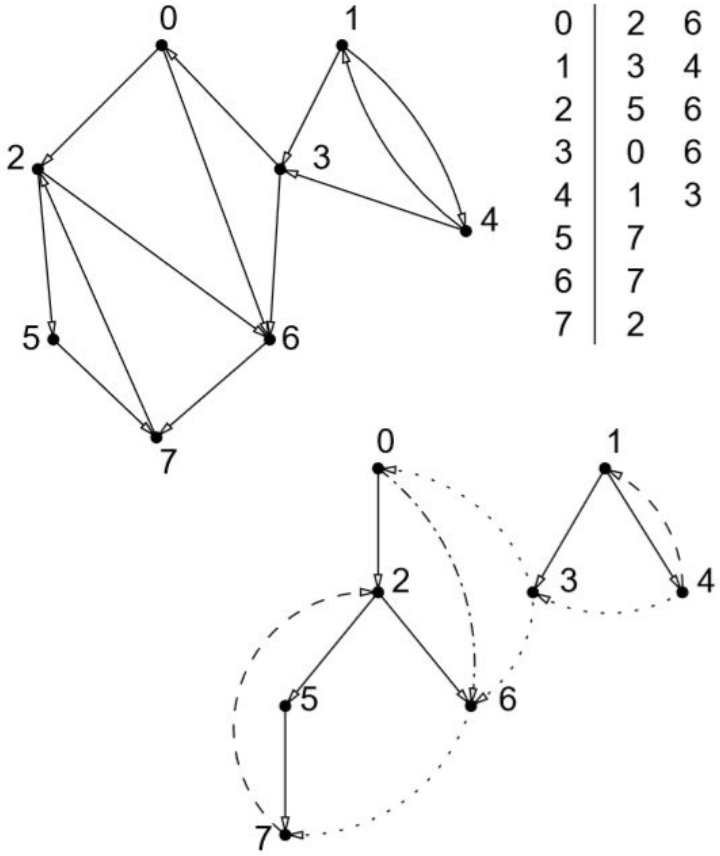
Mergesort kent ook verschillende implementaties. De algemene methode maakt gebruik van recursie.

2. Leg de minst efficiënte methode om een heap op te bouwen uit. En geef zijn efficiëntie.
Reeks 1 vraag 1 => Elementen één per één toevoegen

3. Leg breedte-eerst zoeken uit.

Bijvraag: Teken de dwarsverbindingen bij een ongerichte graaf.
Overlopen van de verschillende niveau's van boven naar beneden en elk niveau van links naar rechts. Dit heet level order overlopen.recursieve methode om de boom te overlopen. De kinderen van elke behandelde knoop mogen slechts behandeld worden na alle resterende knopen op hetzelfde niveau, en voor de kinderen van die knopen. Als we telkens de kinderen van een behandelde knoop in een wachtrij opslaan, en de volgende te behandelen knoop uit die wachtrij halen, dan respecteren we de volgorde.
Bijvraag : Een verbinding met een zwarte knoop die geen opvolger is, heet een dwarsverbinding (cross edge). Die knoop ligt ofwel in dezelfde boom (in een nevendeelboom), of in een andere boom van het bos. Aangezien men de diepte-eerst bomen gewoonlijk van links naar rechts 'tekent', net zoals de boomtakken binnen dezelfde boom, wijzen dwarsverbindingen steeds van rechts naar links.

Figuur 8.8 toont een gerichte graaf, zijn burenlijsten en zijn diepte-eerst bos, met boomtakken (volle lijn), terugverbindingen (streeplijn), heenverbindingen (punt-streeplijn) en dwarsverbindingen (stippellijn). Boomtakken en terugverbindingen herkennen is



Reeks 4

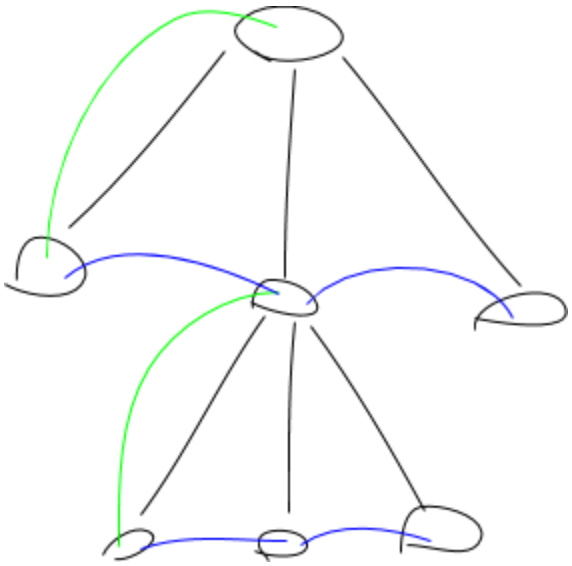
1 Binaire bomen overlopen hoe? En efficiëntie en verband boom en bin boom

Binaire bomen kunnen op twee manieren overlopen worden.

1) *Diepte eerst* gaat zo diep mogelijk in de linkse deelboom, er kunnen 3 gevallen onderscheden worden:
    a) Overlopen in preorder. Eerst wordt de wortel behandelt, dan de linkse deelboom en dan de rechtse deelboom
    b) Overlopen in inorder. Eerst wordt de linkse deelboom behandelt, dan de wortel en dan de rechtse deelboom
    c) Overlopen in postorder. Eerst wordt de linkse deemboom behandelt, dan de rechtse deelbom en dan de wortel

    todo
2) *Breedte eerst* kan geen gebruik maken van de recursieve structuur van de boom. Er moet dus extra bijgehouden of dat een bepaalde knoop al bijgehouden is. Belangrijk bij breedte eerst zoeken is dat een niveau volledig afgewerkt moet worden vooraleer het volgende niveau mag behandelt worden

Een binaire boom is geschikt om d-meerwegsbomen met $d > 2$ voor te stellen. Een binaire boom houdt normaal expliciet zijn linker en rechterkind bij via pointers. Bij d-meerswegsbomen moet de kinderen voorgesteld worden via een gelinkte lijst. Je kan een d-meerwegsboom aanpassen zodat elke knoop een pointer heeft naar zijn eerste kind(groene lijn), en een pointer naar zijn opvolger op hetzelfde niveau(blauwe lijn). Op die manier bootst een d-meerwegsboom een binaire boom na dunno of dat hij dit bedoelt met “verband binaire boom en boom”



2 graaf diepte eerst perf, uitleg en gebruik

Diepte eerst zoeken bij een graaf start vanuit een willekeurige knoop (vrij te kiezen). Eerst moeten alle knopen van de graaf ingesteld worden als “nog niet ontdekt”. Daarna worden de buren “zo links mogelijk” behandeld. Dit is wat dubbelzinnig aangezien een graaf geen boom is, maar zolang een nieuwe knoop een buur heeft die nog niet ontdekt is, zullen eerst deze knopen behandelt worden.

Het instellen van alle knopen op “nog niet ontdekt” is $\Theta(n)$. Elke knoop wordt eenmaal ontdekt en eenmaal behandeld, wat ook $\Theta(n)$ operaties oplevert.

Verder speelt de gebruikte voorstelling van de graaf ook een rol.

- *Burenlijst*: Indien een burenlijst gebruikt wordt is het maar een kwestie van de lijst van buren op te vragen voor een knoop. Zo een lijst heeft lengte m met $m \leq n$. De burenlijst van een knoop overlopen is dus $\Theta(m)$, wat ervoor zorgt dat de totale performantie gelijk is aan $\Theta(n + m)$
- *Burenmatrix*: Indien een burenmatrix gebruikt wordt moeten per knoop altijd alle andere knopen doorlopen worden om na te gaan wie een buur is van die knoop. Elke knoop moet dus *n* vergelijkingen uitvoeren, wat ervoor zorgt dat de totale performantie gelijk is aan $\Theta(n^2)$

Dunno about gebruik

3 quicksort hoe perf verbeteren? En wat is de performantie dan? Vergelijk formules en uitleg

In het normale geval is de performantie van quicksort $O(nlog_2 n)$. Dit kan verbeterd worden indien de recursie gestopt wordt als de deeltabel klein genoeg geworden is, en ze te rangschikken met Insertion Sort.

Reeks 5

1. Performantie van mergesort op n gegevens? Verklaar.

Reeks 3 vraag 1

2. Bespreek verwijderen van sleutels bij binaire zoekbomen en threaded trees. Efficientie als boom n gegevens bevat. Verklaar.

Reeks 8 vraag 2

3. Bespreek efficiente union-find operaties op disjuncte deelverzamelingen. Wat is de efficientie? Verklaar.

Om te testen of twee elementen tot dezelfde deelverzameling behoren, kiest men een willekeurig element uit elke deelverzameling als haar vertegenwoordiger. Een zogenaamde find-operatie op een element vindt de vertegenwoordiger van de deelverzameling waartoe het behoort, zodat twee find-operaties volstaan om na te gaan of twee elementen equivalent zijn. Het samenvoegen van de elementen uit twee deelverzamelingen noemt men een union-operatie. Na elke union-operatie is er een deelverzameling minder, zodat er hoogstens n−1 union-operaties kunnen gebeuren.

Find-operatie:

Aangezien we die elementen niet kennen, moeten we heel de tabel overlopen, zodat de n − 1 mogelijke union-operaties samen Θ(n²) bewerkingen vereisen.Om dat te verbeteren houden we ook nog de elementen van elke verzameling bij in een gelinkte lijst. Die lijsten komen in een tabel, ge¨ındexeerd met de vertegenwoordiger van de deelverzamelingen. Samenvoegen vereist dan enkel de concatenatie van twee lijsten, en het aanpassen van de find-tabel voor de elementen uit een van die lijsten. Toch blijft een performantie van O(n2) mogelijk, bijvoorbeeld door een telkens langer wordende lijst stelselmatig bij een lijst met slechts een element te voegen.

Men kan dit vermijden door zorgvuldiger te werk te gaan bij de union-operatie. De union-by-size heuristiek bijvoorbeeld voegt steeds de kortste lijst toe aan de langere. Daartoe houdt men de lengte van elke lijst bij, en het is triviaal om die aan te passen. Aangezien elk element nu hoogstens⌈lgn⌉keer van verzameling kan veranderen (want telkens wordt zijn verzameling minstens tweemaal zo groot), vereisen n − 1 union-operaties samen nog slechts O(nlgn) bewerkingen. Een reeks van m operaties (initialisatie, find- en union-operaties) heeft dan in het slechtste geval een performantie van O(m + nlgn).

Union-operatie:

De union-operatie kan zeer effici¨ent gemaakt worden door de deelverzamelingen voor te stellen als bomen van een bos. De wortel is de vertegenwoordiger van elke deelverzameling. Omdat elk element zijn vertegenwoordiger moet kunnen terugvinden, maar niet omgekeerd, verwijst elke boomknoop enkel naar zijn ouder. De union-operatie is nu zeer eenvoudig: de wortel van de ene boom wordt een kind van de wortel van de andere.

De performantie kan echter fel verbeterd worden door twee heuristieken te gebruiken:

- De eerste is dezelfde union-by-size heuristiek als bij de effici¨ente find-operatie met gelinkte lijsten, en houdt het aantal elementen per boom bij. Door telkens de boom met minder elementen samen te voegen met deze met meer elementen, beperken we de hoogte van de bomen tot⌈lgn⌉. (Om dezelfde redenen als vroeger. Een element kan hoogstens ⌈lgn⌉ maal van boom veranderen, en telkens neemt zijn diepte met ´ e´en toe, beginnend vanaf nul, als wortel van zijn eigen boom.) De find-operatie wordt dus O(lgn), en een reeks van m operaties vraagt in het slechtste geval O(mlgn) bewerkingen (de O(n)-termen kunnen immers verwaarloosd worden). Als er veel meer find- dan union-operaties zijn dan is dit slechter dan de beste versie van de effici¨ente find. Men kan aantonen dat het gemiddeld geval O(m) bewerkingen vereist, wat dan neerkomt op een gemiddelde geamortiseerde performantie van O(1) per operatie.

- De find-operatie zou optimaal zijn mocht elke knoop rechtstreeks naar de wortel van zijn boom wijzen. Dat zou echter de union-operatie ineffici¨ent maken, omdat daarbij dan alle knopen van ´ e´en van de bomen moeten aangepast worden. Toch kan men dit ideaal benaderen door bij elke find-operatie alle knopen op de gevolgde weg naar de wortel te laten wijzen. (Dat vereist natuurlijk dat die weg tweemaal doorlopen wordt.) Een volgende find-operatie op een van die knopen zal dus optimaal zijn. (En kan ook nog voor andere knopen verbeterd zijn.) Hoe meer find-operaties er gebeuren, des te kleiner wordt de hoogte van de bomen. Deze heuristiek heet path compression, en kan samen met de eerste toegepast worden, omdat hij het aantal elementen in de bomen niet wijzigt.

De performantieanalyse van deze union/find-operaties is uiterst moeilijk, en zelfs nog niet volledig doorgrond. Zo is het niet duidelijk of de tweede heuristiek de gemiddelde performantie van de eerste verbetert. Wat men echter wel weet is dat de performantie in het slechtste geval sterk verbetert door beide heuristieken samen te gebruiken. Voor een reeks van m bewerkingen (gesteld dat $m \geq n$) is de performantie immers $\Theta(m\alpha(m,n))$, waarbij $\alpha(m,n) \leq 4$ voor alle praktische waarden van n en m

Reeks 6

1. Heapsort performantie, formules + uitleg + waarom is heapconstructie toevoegen minder performant dan samenvoegen

Reeks 1 Vraag 1 & Reeks 2 Vraag 1

2. Dijkstra uitleggen + performantie bespreken

Het algoritme van Dijkstra berekent het kortste pad van een startknoop naar een eindknoop in een gewogen graaf met positieve gewichten. Het algoritme vertrekt vanuit de startknoop en overloopt al zijn buren en berekent de kost van de startknoop tot aan elk van zijn buren. Het algoritme bezoekt dan de knoop met het kortste pad. De buren van deze knoop worden ook ontdekt en de kost van de tweede knoop naar elk van zijn buren moet opgeteld worden met de kost van de startknoop naar de tweede knoop. In elk geval blijft het algoritme naar de knoop gaan die het minst "kost".

Aangezien dit algoritme altijd het minimum van de verbindingen neemt, is een prioriteitswachtrij geïmplementeerd met een heap ideaal. (Het origineel algoritme van Dijkstra gebruikte een normale tabel waarin dus lineair moest gezocht worden, wat enkel nuttig is bij zeer dichte grafen en met een burenmatrix ipv burenlijst). Stel *n* het aantal knopen en *m* het aantal verbindingen van een knoop, dan is de performantie $O(mlog_2n)$ want elke knoop kan slechts eenmaal toegevoegd en verwijderd worden uit de prioriteitswachtrij wat $O(nlog_2n)$ is (door heapstructuur), en verder kan de prioriteit van elke knoop hoogstens *m* keer veranderen wat $O(mlog_2n)$ is wat in totaal $O((n + m)log_2)$ oplevert. Aangezien *m* = $O(n)$ doordat de graaf samenhanged is, wordt de performantie dus $O(mlog_2n)$.

3. In detail hashtabellen met chaining uitleggen. Is er ook een andere soort chaining en wat kan je er me over zeggen?

Het is mogelijk dat een bepaalde hashfunctie voor duplicaten zal zorgen. Er moet dus een manier zijn om deze collisies op te vangen bij het hashen van elementen. Eén van die methodes is chaining. Er bestaan hier twee vormen van.

1. *Seperate Chaining*: Elk element in de hashtabel is nu een gelinkte lijst. Wanneer er duplicate elementen op treden worden deze vooraan de gelinkte lijst op de juiste index in de hashtabel toegevoegd. De woordenboekoperaties zijn gedefinieerd als volgt
    a. *Toevoegen*: Een element toevoegen gebeurd vooraan in de gelinkte lijst. Dit is dus $O(1)$
    b. *Zoeken:* In het slechtste geval worden alle elementen in dezelfde gelinkte lijst gestoken. In dit geval is zoeken lineair en is dus $O(n)$. Het gemiddelde geval is iets complexer. De perfecte hashfunctie zou er immers voor zorgen dat elke gelinkte lijst hoogstens één element bevat. Een belangrijke term is hier de *load factor* α (bezettingsgraad). Dit is de verhouding tussen het totaal aantal elementen en het aantal tabelindices (dus $\frac{n}{m}$). Elke gelinkte lijst heeft dus

gemiddelde lengte α en de performantie zou dus normaal $\Theta(\alpha)$ zijn, aangezien hoogstens elk element moet overlopen worden om een ander element te zoeken. Er komt echter nog een constate tijd bij voor het evauleren van de hashfunctie, die niet verwaarloosbaar is met α. De performantie wordt $\Theta(1 + \alpha)$

c. *Verwijderen*: Verwijderen van een gevonden element is $O(1)$

2. *Coalesced Chaining*: Seperate Chaining maakt gebruik van lijsten, Coalesced Chaining maakt gebruik van lijstknopen. Tabel van lijstknopen. Een tabel die op elke plaats niet alleen een waarde kan opslaan, maar ook een verwijzing naar een ander element van de tabel. Een plaats die opgevuld werd bij een collisie kan deel uitmaken van verschillende lijsten. Hierdoor gaan de lijsten samenklitten (coalesced). Coalesced chaining heeft een aantal varianten. Zo kan men het samenklitten van de lijsten uitstellen door de hashtabel onder te verdelen in een adreszone en een berging. De hashfunctie genereert dan uitsluitend indices in de adreszone, en bij een conflict wordt een knoop uit de berging gebruikt om de lijst te verlengen.Standaard in zonder berging waardoor de lijsten vroeger gaan samenklitten.

Reeks 7

1) efficiëntie van quicksort met beste en slechtste geval uitleggen

In het beste geval zal quicksort de tabellen in telkens even grote delen opsplitsen. Beschouw *n* een macht van twee dan geldt:

$$T(n) = cn + 2T(n/2)$$

Dit is een logaritmische afschatting waaruit volgt dat $T(n) = O(nlog_2 n)$. Indien *n* geen macht van twee is geldt dit bij benadering. De term *cn* stelt de tijd voor om de tabellen samen te voegen, wat aanzienlijk kleiner is dan bij andere $O(nlog_2 n)$ methodes (bv mergesort), zodat quicksort het snelst bekende algoritme wordt om willekeurige elemente te rangschikken. Gelijke deeltabellen zijn wel niet altijd realiseerbaar. Een constante grootteverhouding tussen de deeltabellen volstaat ook aangezien ????

In het slechtste geval bestaat één van de twee delen uit slechts één element. Als dit op elk niveau gebeurd blijkt de methode niet sneller te zijn dan Insertion Sort want nu geldt het volgende:

$$T(n) = cn + T(1) + T(n-1)$$

En als dit op elk niveau gebeurd:

$$T(n-1) = c(n-1) + T(1) + T(n-2)$$
$$T(n-1) = c(n-1) + T(1) + T(n-2)$$
$$\vdots$$
$$T(2) = c(2) + T(1) + T(1)$$

Bij het optellen van de gelijkheden:

$$T(n) = c(n + (n-1) + \ldots + 3 + 2) + nT(1)$$

Aangezien T(1) constant is kunnen we dit verwaarlozen, maar de term tussen de haakjes is een rekenkundige reeks, en die is evenredig met n^2, zodat quicksort in het slechtste geval $O(n^2)$ wordt.

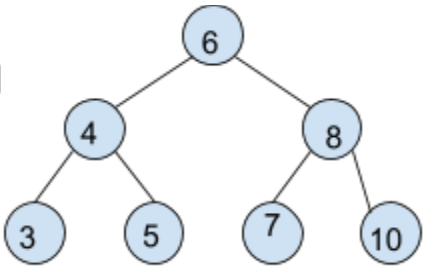
2) leg uit "zoeken in een gesorteerde tabel" en wat is de efficiëntie van dit algoritme

Er bestaan 3 manieren om te zoeken in een gesorteerde tabel

1) *Lineair of sequentieel zoeken* Deze methode zal beginnen vanaf de start van een tabel en zal elk element overlopen totdat het element gevonden is, of als er een element gevonden is dat groter is dan het te zoeken element. Deze methode is echter van weinig nut want dit is $O(n)$ misschien zelfs niet vermelden tijdens een examen, dit is basically gewoon van links naar rechts zoeken
2) *Binair zoeken* Binair zoeken is een efficiënte zoekmethode bij gesorteerde tabellen. Binair zoeken begint in het midden van een tabel en vergelijkt de sleutel (= *a)* op die index met de te zoeken sleutel (= *x)*. Indien *x* kleiner is als *a* dan beschouw je enkel nog deeltabel links van *a*. Indien *x* groter is als *a* dan beschouw je enkel nog de

rechterdeeltabel van *a*. In beide gevallen zet je *a* terug gelijk op het midden van de deeltabel en voer je dezelfde procedure uit totdat je de sleutel gevonden hebt. De performantie van binair zoeken is $O(log_2 n)$. Aangezien de tabel altijd in twee gesplitst wordt halveert het originele probleem bij elke sleutelvergelijking. Dit kan voorgesteld worden als een binaire boom waarbij het linkerkind het middelste element van de linkerdeeltabel bevat en het rechterkind het middelste element van de rechterdeeltabel bevat. Er kunnen dus maar hoogstens $log_2 n$ vergelijkingen plaatsvinden. BINAIR ZOEKEN GEBRUIKT GEEN BINAIRE BOOM, GEWOON TER ILLUSTRATIE

Bv zoek getal 3
In tabel : [3, 4, 5, 6, 7, 8, 10]



Interpolerend zoeken. Deze methode maakt gebruik van de waarschijnlijkheid dat de sleutel op een bepaalde plaats zal zitten. Dit gebruikt quasi dezelfde methode als binair zoeken, behalve de manier waarop het "middelste" element berekent wordt. Bij binair zoeken is dit $\frac{1}{2}(r - l) + r$. Interpolerend zoeken vervangt $\frac{1}{2}$ door $\frac{s - v[l]}{v[r] - v[l]}$. Er kan aangetoond worden (geen bewijs) dat de performantie nu gemiddeld $O(log_2 log_2 n)$ is.

3) dijkstra uitleggen ook met efficiëntie

Vereiste is dat de gewichten positief zijn. Het algoritme steunt op een soortgelijke eigenschap als bij minimale overspannende bomen, en kan de kortste afstanden (en meteen ook de wegen) verbinding per verbinding construeren, zonder op zijn stappen te moeten terugkeren. Het is dus weer een standaardvoorbeeld van een inhalig algoritme. De kortste wegen vanuit het vertrekpunt vormen eens te meer een (overspannende) boom van de graaf. Het algoritme bouwt die boom geleidelijk op, door weer drie groepen knopen te onderscheiden: deze die reeds deel uitmaken van de (parti¨ele) boom (zwart), de randknopen die rechtstreekse buren zijn van een knoop uit de boom (grijs), en de rest (wit). Voor de randknopen wordt nu de (voorlopige) kortste afstand vanuit het vertrekpunt bijgehouden (via een weg van boomknopen). Telkens wordt de randknoop met de kleinste voorlopige afstand geselecteerd, en in de boom opgenomen. Met deze nieuwe definitieve afstand wordt de rand zo nodig aangepast. Dit gaat door tot alle knopen in de boom zitten.
Wanneer er een nieuwe knoop in de boom wordt opgenomen, onderzoekt men al zijn buren. Is een buur reeds een randknoop, dan kan de weg via de nieuwe knoop korter zijn, en moet de voorlopige afstand van die buur aangepast worden. Een buur kan via de nieuwe knoop voor het eerst verbonden worden met de boom, en krijgt een voorlopige beginafstand. Met een buur die

reeds in de boom zit tenslotte moet er niets gebeuren, want de kortste afstand van die buur was reeds bekend, en de afstand van de nieuwe knoop via die buur is zeker niet korter.
Opnieuw moeten we randknopen met afstanden bijhouden, zodat de knoop met de kleinste afstand effici¨ent verwijderd kan worden. Een goede implementatie gebruikt dus een prioriteitswachtrij, net zoals het algoritme van Prim, en met dezelfde voorzieningen om de prioriteit van de knopen effici¨ent te kunnen aanpassen. De performantie is dus ook O(mlgn).

De originele versie van Dijkstra (1959) gebruikte geen prioriteitswachtrij, maar een gewone tabel (zoals de originele methode van Prim), die dan sequentieel werd overlopen. Ook hier werd zoeken naar het volgende minimum gecombineerd met aanpassen van de afstanden. De performantie is dus weer Θ(n2), wat nu nog enkel zin heeft voor dichte grafen.

Reeks 8

P: performantie van gewone quicksort, werk berekening uit voor gemiddeld geval.
Bijvraag: hoe performantie verbeteren?
Reeks 7 Vraag 1 (beste en slechtste geval).

Het gemiddelde geval doet zich voor wanneer de pivot met dezelfde kans elk element kan zijn met als gevolg dat elke grootte van deeltabellen even waarschijnlijk is. Dit komt neer op volgende formule:

$$T(n) = cn + \frac{1}{n}\sum_{i=1}^{n}(T(i-1)+T(n-i))$$

Aangezien de meeste termen tweemaal in de som voorkomen schrijven we

$$T(n) = cn + \frac{2}{n}\sum_{i=0}^{n-1}T(i)$$

Op blad van "tekennenstof" staat dat we enkel de formule moeten kunnen opstellen, dus hier stoppen

Performantie verbeteren: Reeks 4 vraag 3

S: hoe verwijder je een element uit bin. zoekboom en threaded boom? Geef performantie.

1. *Element verwijderen uit binaire zoekboom*. Een element (= knoop) verwijderen uit een binaire zoekboom kan onderverdeeld worden in 3 gevallen.
    a. *De knoop heeft geen kinderen*: Indien een knoop geen kinderen heeft kan het element normaal verwijdert worden zonder dat de eigenschap van een binaire zoekboom verstoort worden.
    b. *De knoop heeft één kind*: Het kind van de knoop moet het kind worden van de ouder van de knoop die verwijdert wordt. Indien de knoop die verwijdert moet worden een linkerkind is van de ouder van die knoop, moet het kidn van de knoop ook een linkerkind worden, en vice versa.
    c. *De knoop heeft twee kinderen*: Wanneer een knoop twee kinderen heeft zoekt men de opvolger van de knoop in de rechterdeelboom. Deze opvolger zal de huidige knoop vervangen. Meestal wisselt men af door de knoop ook te vervangen door de voorloper in de linkerdeelboom, zodat de boom niet onevenwichtig wordt.
2. *Element verwijderen uit threaded tree.* Dezelfde gevallen gelden als bij een binaire zoekboom. Er moet wel altijd rekening gehouden dat de threads correct blijven

Peformantie is in beide gevallen $O(log_2 n)$ aangezien eerst de knoop moet gezocht worden, en dan eventueel de opvolger/voorloper, wat dus nooit meer operaties zal opleveren als de hoogte van de boom.

G: bespreek Kruskal.

Dit algoritme bouwt een MOB op uit afzonderlijke deelbomen, die geleidelijk met elkaar verbonden worden tot er maar 1 boom meer overblijft. De lichtste graafverbinding die twee van de deelbomen verbindt, is de lichtste verbinding uit de snede gevormd door de knopen van een van die deelbomen, en alle andere knopen. Daarom worden de graafverbindingen volgens stijgend gewicht onderzocht, en enkel als ze twee deelbomen verbinden worden ze toegevoegd. Prioriteitswachtrij gebruiken, want de resterende kleinste verbinding hebben we nodig (niet op voorhand rangschikken van de verbindingen).

De opbouw van de heap is O(m). in het slechtste geval heb je elke verbinding nodig = O(mlgm). Voor elke verbinding is er 1 union en 2 find methodes nodig = O(m + n). Totaal = O(mlgm).

Reeks 9

- mergesort performantie berekenen en uitleggen

Reeks 3 Vraag 1

- methodes van open adressering bespreken + performantie (zonder berekening)

Er bestaan drie methodes van open adressering.

1. *Lineair Testen*. Indien een sleutel al reeds bestaat in de hashtabel, dan wordt er telkens één plaats opgeschoven (module de grootte van de hashtabel) tot er een plaats vrij is. Dit is vrij eenvoudig maar kan leiden tot primaire clustering. Dit wil zeggen dat er vrij grote clusters zullen onstaan van sleutels die allemaal naast elkaar liggen, wat de performantie negatief beïnvloedt.
2. *Kwadratisch Testen*. Deze zoeksequentie maakt gebruik van de formule $(h(s) + c_1 i + c_2 i^2) \bmod m$ waarbij h(s) een hashfunctie is en c_1, c_2 constanten zijn. Primaire clustering wordt hier vervangen door secundaire clustering
3. *Dubbele Hashing*. Deze methode maakt gebruik van twee *verschillende* hashfuncties.

- constructie-eigenschap van minimale overspannende bomen uitleggen en aantonen.
Bijvraag: hoe gebruiken Prim en Kruskal deze eigenschap?

Een overspannende boom (‘spanning tree’) van een ongerichte (samenhangende) graaf is een (wortelloze) boom met dezelfde knopen en met een deel van de verbindingen als takken. Zowel diepte-eerst als breedte-eerst zoeken genereren overspannende bomen, en het is duidelijk dat een graaf meerdere overspannende bomen kan hebben. Bij een gewogen graaf definieert men het gewicht van een overspannende boom als de som van de gewichten van zijn takken, en een minimale overspannende boom (MOB) heeft dan het kleinste gewicht van alle dergelijke bomen.1 Ook een MOB is niet noodzakelijk uniek.

Alle efficiente algoritmen construeren een MOB tak per tak. Daarbij maken ze telkens een lokaal optimale keuze. Het eindresultaat is echter een globaal optimum. Dat is niet vanzelfsprekend: voor veel problemen vindt deze werkwijze enkel een suboptimale oplossing. Algoritmen die (globaal) optimale oplossingen vinden, door steeds lokaal optimale keuzes te maken (zonder ooit op hun stappen te moeten terugkeren), noemt men ‘inhalig’ (greedy).

Om die nieuwe verbinding te kiezen, kunnen we de volgende algemene eigenschap gebruiken (Tarjan, 1983), die gebruikt maakt van het begrip snede. Een snede S van een samenhangende graaf is een verzameling verbindingen die de knopenverzameling in twee niet-ledige stukken A en B verdeelt. Dit houdt in dat een verbinding (i,j) in S zit als en slechts als $i \in A$ en $j \in B$ of, omgekeerd, $j \in A$ en $i \in B$.

Algoritmes verschillen op de manier waarop ze de sneden kiezen

PRIM = telkens uitbreiden met de lichtste verbinding uit S

KRUSKAL = de lichtste verbinding die twee van de deelbomen verbindt, is de lichtste verbinding uit de snede gevormd door de knopen van een van die deelbomen, en alle andere knopen.

Reeks 10

Vraag 1: Men kan ervoor zorgen dat quicksort bijna zeker efficiënt is. Hoe doet men dat, en wat is die performantie? Toon aan waarom. (Geef niet enkel formules. Verklaar ze ook.)

De eerste manier is om het spilelement zo te nemen dat het de mediaan is van 3 waarden. Meestal het linkse, het middelste en het rechtse element van de tabel. Dit vermijdt met grote kans dat het spilelement de grootste of kleinste waarde is in de tabel (en dus deeltabellen vermijdt met grootte 1) want dan moeten 2 van de 3 elementen gelijk zijn aan het maximum of minimum. In het geval dat het spilelement toch het maximum of minimum is kan dit geen kwaad want om $O(n^2)$ gedrag te vertonen moet dat bovendien bij veel partities gebeuren. Die kans is klein.

Andere methoden? Misschien Reeks 4 Vraag 3?

Vraag 2: Bespreek het algoritme van Dijkstra. Hoe efficiënt is dat algoritme? Verklaar.

Reeks 7 vraag 3

Vraag 3: Bespreek toevoegen van sleutels bij binaire zoekbomen, en ook bij 'threaded trees'. Wat is de efficiëntie van deze operatie, als de bomen n knopen bevatten? Verklaar (zonder bewijs).

1. *Toevoegen van een element aan een binaire zoekboom*: Toevoegen aan een binaire zoekboom is heel eenvoudig indien er veronderstelt wordt dat er geen duplicate waarden zijn. Het meeste werk wordt gedaan door een zoek operatie. Deze operatie voert het standaard zoek-mechanisme uit. Wanneer het te zoeken element kleiner is als een knoop, dan gaan we naar de linkse deelboom. Andersom gaan we dan naar de rechterdeelboom. Deze operatie geeft een pointer terug naar een knoop die opgevuld moet worden met het nieuwe element. Bij een binaire zoekboom kan er nooit tussengevoegd worden.

   Wanneer duplicaten wel toegelaten worden, zijn er 3 mechanismen om dit te implementeren.
   1. Elke knoop bevat een gelinkte lijst met daarin duplicate waarden. Dit is de eenvoudigste oplossing maar qua geheugengebruik is dit niet optimaal
   2. Elke knoop houdt een extra bit bij (meestal een byte omdat de meeste programmeertalen niet toelaten om individuele bits aan te passen) dat zegt of de volgende duplicate waarde een linkerkind of een rechterkind moet worden. Wanneer dit gedaan wordt moet deze bit wel geïnverteerd worden. Het nadeel hierbij is dat er bij elke knoop dus een extra bit (byte dus) bijgehouden moet worden
   3. In plaats van een extra byte bij te houden wordt er random gegenereerd waar de knoop moet komen. Een goede randomgenerator kan er dus voor zorgen dat

statistisch 50% van de gevallen links komt, en 50% van de andere gevallen rechts.

De performantie hangt af van de zoek-operatie en is dus $O(log_2 n)$.

2. *Toevoegen van een element aan een threaded tree*. Basically hetzelfde BEHALVE: Bij het toevoegen moet de voorloper en de opvolger nog bepaald worden. Indien de nieuwe knoop een linkerkind is wordt de opvolger van de nieuwe knoop gewoon de ouder, De voorloper van de nieuwe knoop was de voorloper van de ouder, die dus als linkerkindpointer bijgehouden werd. Indien het kind een rechterkind wordt is hetzelfde maar de voorloper is dan de ouder, en de opvolger was de opvolger van de ouderknoop.

   Performantie is ook $O(log_2 n)$

Reeks 11

1. Binaire bomen overlopen hoe? Implementatie en efficiëntie en verband boom en bin boom

Reeks 4 vraag 1

2. Leg breedte-eerst zoeken uit bij graaf, performantie, waar gebruikt

Breedte-eerst zoeken begint vanaf een willekeurige knoop (vrij te kiezen). Alle buren van deze knoop worden in een wachtrij geplaatst. Deze wachtrij bepaalt de volgorde van overlopen. Wanneer een knoop van de wachtrij wordt gehaald en dus behandelt wordt, worden zijn buren die nog niet ontdekt zijn (en bij een gerichte graaf, enkel de buren naar waar de knoop naartoe kan gaan) aan de wachtrij toegevoegd.

De initialisatie van breedte eerst zoeken (alle knopen op 'niet ontdekt' zetten) is $\Theta(n)$.
Elementen op een wachtrij zetten en afhalen is $O(1)$, voor alle knopen samen is dit dus $\Theta(n)$
Verder speelt de gebruikte voorstelling van de graaf ook een rol.

- *Burenlijst*: Indien een burenlijst gebruikt wordt is het maar een kwestie van de lijst van buren op te vragen voor een knoop. Zo een lijst heeft lengte m met $m \leq n$. De burenlijst van een knoop overlopen is dus $\Theta(m)$, wat ervoor zorgt dat de totale performantie gelijk is aan $\Theta(n + m)$
- *Burenmatrix*: Indien een burenmatrix gebruikt wordt moeten per knoop altijd alle andere knopen doorlopen worden om na te gaan wie een buur is van die knoop. Elke knoop moet dus *n* vergelijkingen uitvoeren, wat ervoor zorgt dat de totale performantie gelijk is aan $\Theta(n^2)$

Dunno "waar gebruikt",

3. Selectie bij n elementen best en worst case

De selectie opdracht wil zeggen: zoek het *k*-de kleinste element van *n* elementen $(1 \leq k \leq n)$.
Geen enkel algoritme kan bepalen wat het *k*-de kleinste element is zonder de k-1 kleinere elementen gevonden te hebben. De selectie operatie levert dus de *k* kleinste elementen op. Er zijn 3 mogelijkheden op basis van de waarde van *k*

1. Wanneer *k* klein is kan gerust **selection sort** gebruikt worden om het *k*-de kleinste (of grootste element te vinden. De operatie is dan $O(kn)$. Dit is wat wij gedaan hebben op de test, maar op de test was *k* "heel groot" en niet "klein"
2. Voor iets grotere *k* wordt gebruik gemaakt van **heap sort.** Voer de *k* eerste stappen uit van heapsort (minheap of maxheap afhankelijk indien je *k*-de grootste of *k*-de kleinste zoekt). Dit geeft een performantie van $O(n + klog_2 n)$
3. Voor heel grote *k* (zoals op de test) wordt gebruik gemaakt van **quick sort**. Aangezien quick sort elke keer deeltabellen maakt waarbij het linkerdeel kleiner is als het spilelement en het rechterdeel groter is als het spilelement (en dus de pivot op zijn gesorteerde plaats terecht komt) zijn er 3 mogelijkheden:

a. Als $k = i$, dan hebben we wat we zoeken
b. Als $k < i$, dan zoek je verder in de linker deeltabel, en dus ook naar het k-de kleinste element
c. Als $k > i$, dan zoek je verder in de rechter deeltabel, maar naar het $(k - i)$-de kleinste element

Efficientie van quick sort bij selectie : slechtste geval : $O(n^2)$
Gemiddeld geval : $O(n)$
Beste geval : $O(n)$

Reeks 12

1) Efficiëntie mergesort (gelijke delen en constante verhouding)
Reeks 3 Vraag 1
2) Kortste afstanden bij gerichte lusloze grafen (hoe + efficiëntie) en verband met projectplanning
Om de kortste afstand te berekenen in een gerichte lusloze graaf moet eerste de graaf toplogisch gerangschikt worden. Dit wil zeggen dat alle verbindingen naar rechts wijzen. Hiervoor zijn er twee mogelijkheden;

1. *Via diepte-eerste zoeken:* Het is belangrijk dat diepte-eerst in postorder wordt afgewerkt. (aangezien het een lusloze gerichte graaf is zie ik het als een soort boom, maar dit wordt niet expliciet vermeld in de cursus). Dus eerst de linkerdeelboom verwerken, dan de rechterdeelboom en dan de wortel. Op die manier werk je dus rechts van links op de topologische as. Aangezien dwarsverbindingen naar links wijzen en heenverbindingen naar onder, zijn alle verbindingen vanuit een knoop behandeld. Aangezien dit diepte-eerst zoeken is heeft het gewoon de performantie van diepte-eerst zoeken en is dit $\Theta(n + m)$ (indien gebruik van burenlijst)
2. *Via ingraden*: De ingraad van een knoop is het aantal inkomende verbindingen en een lusloze graaf heeft minstens één knoop met ingraad 0. Die knoop kan als eerste geplaatst worden en dan kan de subgraaf beschouwd worden zonder deze knoop. De ingraad van alle knopen dat de oorspronkelijke knoop als inkomende verbinding had, moet dus niet meer beschouwd worden (dus ingraad verlagen met 1). Er zal altijd een knoop zijn met ingraad 0. Als datastructuur kan een stapel of wachtrij gebruikt worden om de knopen op te plaatsen.

   Elke knoop wordt eenmaal toegevoegd en verwijderd van de stapel of wachtrij en is $\Theta(n)$. Ook moet elke buur van elke knoop behandeld worden en dt is $\Theta(m)$. In totaal is de performantie $\Theta(m + n)$

Projectplanning bestaat uit een aantal taken die moeten uitgevoerd worden om een bepaalde doelstelling te halen. Sommige taken vereisen dat er andere taken eerst uitgevoerd moeten worden (bv eerst brug ontwerpen en dan pas bouwen hopelijk)
3) Hoe zoeken in geordende tabellen + efficiëntie
Reeks 7 Vraag 2

Reeks 13

1) Bespreek de performantie van open adressering bij hashing.
Opnieuw wordt de performantie volledig bepaald door de bezettingsgraad α (die hier, zoals gezegd, nooit groter is dan 1):

- Hoeveel testen zijn er nodig om een afwezige sleutel te zoeken?

Er gebeurt altijd 1 test, de 2de test zal pas gebeuren als de eerste een bezette plaats vind. Enz voor de verdere testen. De 2de test zijn er nog n-1 plaatsen bezet en m-1 resterende plaatsen enzovoort. Het gemiddeld aantal testen voor een afwezige sleutel is dan : 1/(1-α). (voor alpha = 0.5 == 2 en voor alpha = 0.9 == 10)

- Hoeveel testen zijn er nodig om een aanwezige sleutel te zoeken?

Hetzelfde zoeksequentie als bij toevoegen en dus nog afwezig was. Dus gemiddeld aantal testen is hoogstens m/(m-i). Het gemiddeld aantal testen voor een aanwezige sleutel is dan : 1/α * ln(1/(1-α)).

2) Bespreek het algoritme van Floyd-Warshall. Performantie? Verklaar.
Werkt met dynamisch programmeren. Het belangrijke is hoe de deelproblemen efficient geordend worden zodat we bottom-up kunnen werken met zo weinig mogelijk geheugen en tijdverbruik. De dichte graaf wordt voorgesteld als een burenmatrix met volgende definities :

$$g_{ij} = \begin{cases} 0 & \text{als } i = j \\ \text{gewicht van verbinding } (i,j) & \text{als } i \neq j \text{ en de verbinding bestaat} \\ \infty & \text{als } i \neq j \text{ en de verbinding niet bestaat} \end{cases}$$

De gezochte oplossing word opgeslaan in een afstandenmatrix. Waarvan elk element de korste afstand bevat. En dan is er nog een 3de voorlopermatrix, waarvan het element Vij de voorloper is van knoop j op de korste weg vanuit knoop i.

- Als k geen intermediare knoop van Wij is, dan is Wij de vorige korste weg die k niet mocht gebruiken.
- Indien wel, dan kunnen we Wij onderverdelen in deelwegen. Deelwegen van een korste weg zijn ook korste wegen. De lengte van beide is dus reeds bekend, zodat we het nieuwe minimum kunnen bepalen.

$$a_{ij}^{(k)} = \min\left(a_{ij}^{(k-1)}, a_{ik}^{(k-1)} + a_{kj}^{(k-1)}\right)$$

De performantie is theta(n³). De voorlopermatrix V kan samen met A opgebouwd worden door een analoge reeks voorlopermatrices te bepalen. De laatste matrix is dan de gezochte V.

3) Bespreek uitwendig samenvoegen bij uitwendig rangschikken (en performantie). Dus niet hoe je tot die sequenties komt (zoiets ongeveer)
uitwendig samenvoegen: => mergesort
Performantie:

Het aantal schijfoperaties bij uitwendig samenvoegen wordt dan

$$\Theta\left(\frac{n}{d}\log_{m/d}\frac{n}{d}\right) = \Theta\left(\frac{n}{d}\frac{\log(n/d)}{\log(m/d)}\right)$$

Dat is echter een factor

$$\frac{\log(n/d)}{\log n}\frac{\log m}{\log(m/d)} \approx \frac{\log m}{\log(m/d)}$$

meer dan het optimum. Voor grotere d wordt die noemer klein, zodat het verschil belangrijk kan zijn.

Uitwendig rangschikken :

- Load and store: telkens zoveel mogelijk gegevens inlezen en rangschikt die met een efficiente methode en schrijft ze weer uit.
- Replacement selection: inwendig geheugen gebruiken als een soort filter waarbij continue gegevens worden ingelezen, gegevens met kleiner waarden worden doorgelaten en in volgorde uitgeschreven. Gegevens met grotere sleutels worden tegelijk tegengehouden. Onthoudt steeds de sleutel van het laatst uitgeschreven element.

Performantie: theta(lg n)

Reeks 14

1) Bespreek dijkstra algemeen en verklaar efficiëntie

Al opgelost

2)Selectieoperatie uitleggen voor kleine, middelgrote en grote k (beste en slechste geval geven voor grote k)

Reeks 11 vraag 3

3)Heap als prioriteitswachtrij, wat zijn de operaties en verklaar de performantie

In de cursus op p 128 staat; "voor de opslag van een collectie gegevens waaruit …… denken we aan een prioriteitswachtrij, geïmplementeerd met een heap". Er staan ook nog heap-operaties op p30, maar die zijn ook van belang denk ik. Operaties 1, 2, 3 staan op p30, operatie 4 staat op p128-129

Een prioriteitswachtrij is een lineaire structuur waarbij zeker vooraan wordt afgenomen, en afhankelijk van de prioriteit van een element ergens in de tabel wordt tussengevoegd. Een maxheap (veronderstel grote prioriteit = hoger number) is een ideale structuur voor het voorstellen van een prioriteitswachtrij aangezien de wortel het element bevat met de hoogste prioriteit. Volgende operaties moeten voorzien zijn:

1. *Toevoegen van een element. (push)*
   Een nieuw element wordt toegevoegd aan een reeds bestaande of ledige heap. Indien de heap al elementen heeft, moet de prioriteit van het nieuwe element vergeleken worden met de prioriteit van zijn ouder. Indien de prioriteit van het nieuwe element groter is moeten de twee knopen omgewisseld worden. Deze operatie blijft zich herhalen totdat het nieuwe element de wortel is, of totdat er een ouder is die een grotere prioriteit heeft als het nieuwe element. Een element toevoegen aan een tabel is $O(1)$. Het nieuwe element moet hoogstens $O(log_2 n)$ maal geswapt worden. Toevoegen is dus $O(log_2 n)$
2. *Het wortelelement vervangen*
   Indien het nieuwe element groter is als de oorspronkelijke wortel, wordt de heapvoorwaarde niet verstoord. Indien het nieuwe element kleiner is als de oorspronkelijke wortel, moet de nieuwe wortel naar beneden dalen totdat de heapvoorwaarde terug voldaan is. Neem telkens het maximum van beide kinderen voor te swappen. Het element vervangen is $O(1)$. Daarna moet de nieuwe wortel nog hoogstens $O(log_2 n)$ geswapt worden naar beneden. Het wortelelement vervangen is dus $O(log_2 n)$
3. *Het wortelelement verwijderen (pop)*
   Het element met de grootste prioriteit afhalen komt overeen met het wortelelement verwijderen. Dit gebeurd op volgende manier. Verwissel het element op de laatste index met de wortel (swap) en verklein de tabel met 1. De heapvoorwaarde is nu wel verbroken. Dit komt neer op het vervangen van een

wortelelement. De eerste swap met de wortel en het laatse element is $O(1)$. De wortel vervangen is $O(log_2 n)$ dus verwijderen is ook $O(log_2 n)$.

4. *Prioriteit van een element aanpassen*
   Om de prioriteit van een element aan te passen zou de plaats van elke knoop in de heap bekend moeten zijn, wat meestal niet het geval is. De heap lineair doorzoeken is niet efficiënt. Er moet dus een extra tabel bijgehouden worden met de posities van elke knoop. Een knoop vinden wordt dan $O(1)$ en wijzigen van de prioriteit wordt $O(log_2 n)$. Een andere mogelijkheid is om de een nieuwe knoop toe te voegen met een andere prioriteit. De prioriteitswachtrij kan dan meerdere exemplaren bevatten van dezelfde knoop. Wanneer het minimum verwijdert wordt moet er getest worden of de knoop al niet in de MOB zit. Deze oplossing gebruikt veel meer geheugen.

Leg backtracking uit. Geef de manieren om deze methode te verbeteren.

- **Ook uitleggen dat er met een boom gewerkt wordt en hoe die ineen zit.**

Het is de bedoeling dat er systematisch te werk wordt gegaan waardoor elke oplossing aan bod komt zonder alles in geneste forlussen te steken. Oplossing is om incrementeel te construeren. Impliciet wordt er bij backtracking gebruik gemaakt van een boom die diepte-eerst zoekt. De knoop waar we ons in bevinden wordt voorgesteld door een kandidaat-deeloplossing. Afdalen in de boom komt overeen met het toevoegen van een element aan de oplossing. De methode kan op zijn stappen terug keren.
Om de performantie van backtracking te verbeteren probeert met te snoeien in de deeloplossingen. Mogelijke verbetertechnieken

- Variable ordenen : volgens stijgend aantal mogelijke waarden
- Waarde ordenen : best toekennen volgens dalend aantal mogelijkheden (makkelijkste waarden eerst)
- Verder terugkeren : niet 1 stap terug nemen , maar terugkeren naar de variabele die de deeloplossing ongeldig maakte
- Vooruit testen : nagaan of er nog minstens 1 mogelijke waarde overblijft voor alle resterende variabelen
- Symmetrieen : deelbomen die dezelfde oplossing opleveren die we reeds gevonden hebben of die er eenvoudig mee verwant zijn, kunnen we buiten beschouwing laten
- Snoeien bij optimisatie : voorlopig beste oplossing aanvaarden.

Combinatorisch probleem = adhv voorwaarden, alle elementen vinden die voldoen aan de voorwaarden
Combinatorisch optimalisatieprobleem = adhv voorwaarden en een evaluatiefunctie, een element vinden die aan alle voorwaarden voldoet en volgens de functie de beste score behaalt.

2) Leg uit hoe we de kortste weg vanuit een knoop kunnen berekenen in een 'DAG'. Leg ook de verschillende mogelijkheden van projectplanning uit bij DAG's.
Bijvraag: waar moeten we starten bij topologisch rangschikken en waarom bekijken we de knopen links ervan niet meer?
Staat hier al ergens maar andere vraagvorm, is gewoon projectplanning

Antwoord: we beginnen bij de wortel en de knopen links ervan moeten we niet bekijken want daar naartoe kunnen we toch geen kortste weg berekenen