1a) **i.** $O(1)$, constante tijd: er hoeven aan het begin (of eind) van de lijst slechts een aantal pointers veranderd.
ii. $O(\ln n)$, logaritmische tijd: vinden van de positie, en ten hoogste één rotatie op het pad, dus ten hoogste evenredig aan de hoogte van de boom.
iii. $O(1)$: bij het toevoegen in een niet overmatig geclusterde hash-tabel kan de positie in een klein aantal stappen gevonden worden,
iv. $O(n)$, lineaire tijd: in het slechtste geval bevat de tabel een groot cluster, dat geheel gevolgd moet worden.

b) Linked lists preserve the sequence of the elements, which is not the case with sets. The elements in BSTs or hash tables are not ordered in a usable way so we cannot find, for example, the first or the next element.

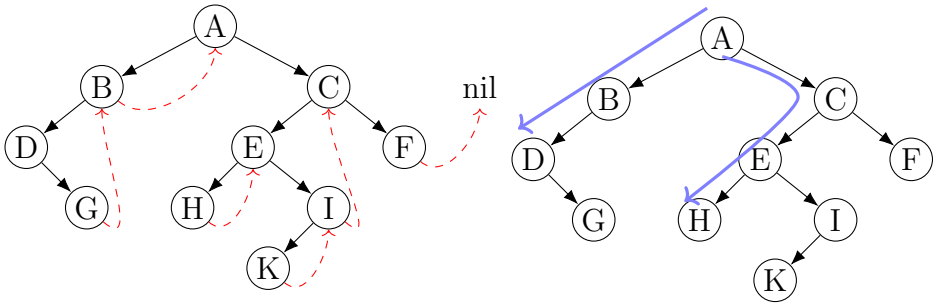
c) In BSTs the elements are sorted, which makes operations like finding the minimum value or values within a range very efficient to compute. Hash tables would be more efficient when searching for specific elements, but not when comparison operations have to be performed.

d) Open addressing: k is stored elsewhere, according to the applied strategy, chained hashing: multiple keys are stored in the same address.

2. Voor de duidelijkheid: inorder = symmetrisch = LWR.

De symmetrische wandeling in het voorbeeld is D, G, B, A, H, E, K, I, C, F.

a) De draden naar de inorder opvolger als in de afbeelding (rood gestippeld). De laatste knoop krijgt een nil-pointer als draad; dat is consistent met het feit dat alle ontbrekende (rechter-)takken naar draad worden omgezet.



b) *Move to first.* De eerste knoop die bezocht moet worden staat helemaal links in de boom, en is vanuit de wortel met een simpele while-loop te vinden. (voorbeeld A → B → D blauw aangegeven in de rechter figuur)

nb. Bij een standaard bedrade boom is in elke knoop aangegeven of de rechter pointer een tak danwel een draad is. De draden zijn al aangebracht, en worden tijdens de wandeling niet verwijderd (dus niet verwarren met Morris).

Move to successor. Algoritme: Als de knoop een (rechter-)draad heeft, dan wijst deze naar de opvolger. Anders, als de knoop een rechtertak heeft, is de opvolger de meest linker knoop van de rechter subboom. (voorbeeld A → C → E → H blauw aangegeven in de figuur)

Done. De knoop is nil (omdat de laatste draad is gevolgd).

c) De knopen waar naartoe draden wijzen, zijn ook de knopen die tijdens de stapelwandeling op de stapel gezet gaan worden. Zij worden gebruikt om later tijdens de wandeling weer terug te kunnen komen hogerop in de boom. Dus, de knopen die op het pad naar de bezochte knoop een linker kind hebben staan op de stapel. Bijvoorbeeld, bij het bezoeken van D (of van G) staan B en A op de stapel, en bij het bezoeken van K, staan I en C op de stapel.

Move to first. Vanuit de wortel weer zo ver mogelijk naar links. In elke stap wordt de knoop op stapel gezet. Dat zijn dus de knopen inclusief de wortel, maar zonder de knoop waar we eindigen.

Move to successor. Als de knoop een rechter kind is, dan is de opvolger op de manier van hiervoor te vinden: rechts, zo ver mogelijk links. Op elk moment dat we links gaan pushen we de knoop op de stapel. Anders, als de knoop geen rechter kind heeft, dan poppen we de opvolger van de stapel.

Done. De stapel is leeg op het moment dat die gepopt zou worden om een opvolger te vinden (bij een knoop zonder rechter kind dus).

Let wel, de knoop waar we eindigen hoeft geen blad te zijn. De laatste knoop heeft weliswaar geen rechter kind, maar zou wel een linker kind kunnen hebben.

3a) Een binairy heap is een (1) complete binaire boom, (2) waarvan de knopen de *heap eigenschap* bezitten: de waarde in elke knoop is groter dan die van zijn kinderen.

Vanwege (1) wordt de boom (3) geïmplementeerd met een array, waarbij de kinderen van een knoop i gevonden kunnen worden op positie $2i, 2i+1$.

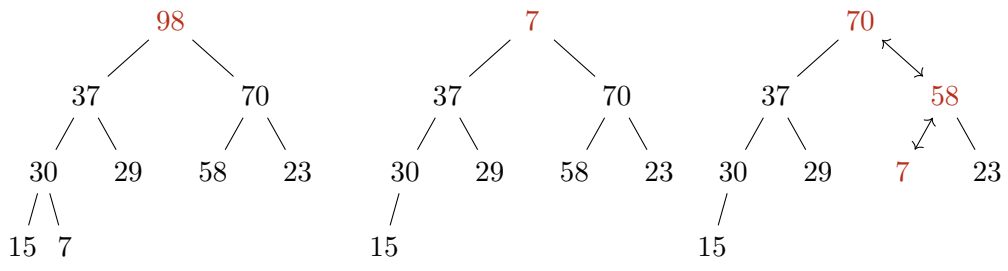
Bij *bubble up* is de waarde van een knoop [meestal een blad] groter dan die van zijn ouder, en wordt de knoop op de juiste plek gebracht door herhaald wisselen met de ouder, totdat weer aan de heap eigenschap is voldaan.

Bij *trickle down* is omgekeerd de waarde in een knoop [bijvoorbeeld de wortel] kleiner dan een van de kinderen, en verwisselen we die waarde herhaald met die van het grootste kind, totdat weer aan de heap eigenschap is voldaan.

oeps. Verwar de heap niet met de *priority queue*, dat is een adt, waarvan de heap een mogelijke implementatie is. Verder is de heap geen 'zoekboom', die hebben een andere ordening tussen de knoopwaarden.

b) De binary heap $[98, 37, 70, 30, 29, 58, 23, 15, 7]$ kan naar keuze direct worden gebruikt als array, of je kunt een boom representatie maken. Ter illustratie geven we allebei.

DeleteMax: Verwijder het maximum, dit staat in de wortel (positie 1) en plaats daar nu het laatste element (van positie 9). Herstel de heap eigenschap met *trickle down*.

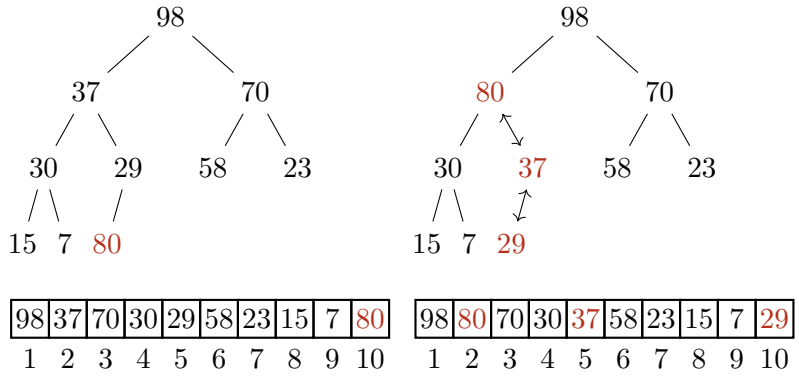


98	37	70	30	29	58	23	15	7
1	2	3	4	5	6	7	8	9

7	37	70	30	29	58	23	15
1	2	3	4	5	6	7	8

70	37	58	30	29	7	23	15
1	2	3	4	5	6	7	8

Voeg 80 toe [aan de oorspronkelijke heap]: Plaats de nieuwe waarde achteraan, op de eerste vrije plek. Herstel de heap eigenschap met *bubble up*.



	98	37	70	30	29	58	23	15	7	80
	1	2	3	4	5	6	7	8	9	10

	98	80	70	30	37	58	23	15	7	29
	1	2	3	4	5	6	7	8	9	10

c) *Heapify* is een operatie die in een ongeordend array de heap structuur aanbrengt. Het werkt achterstevoren: vanaf het laatste element worden de elementen toegevoegd aan onderliggende elementen met trickle down.

Twee heaps samenvoegen kan door ze in één array te kopieren en dan heapify uit te voeren.

oeps. Wanneer we de elementen uit de ene heap aan de andere toevoegen met deleteMax [uit de eerste heap] en/of insert [naar de andere] wordt de complexiteit $O(n \lg n)$, dus meer dan lineair, omdat elke afzonderlijke heap operatie $O(\lg n)$ kost.

Bij *leftist heaps* kunnen twee heaps samengevoegd worden (en zelfs in logaritmische tijd) met de operatie Zip, maar deze operatie is niet geschikt voor de binaire heap. De ene implementatie is met een array, de andere gebruikt pointers.

4a) Initialiseer `conn[i,j]` op `true` als (i, j) een tak is, en anders `false`.

Vervang de `if .. fi` binnen de drie loops door

```
if conn[i,k] and conn[k,j] then conn[i,j] = true fi
```

b) Het geheim zit in de waarde k. Als tijdens het algoritme de afstand korter wordt voor een `dist[i,j]` dan weten we dat k op het kortste pad van i naar j ligt. Tenzij bij een latere k de afstand weer verkort wordt! Er zijn twee mogelijke oplossingen.

(i) Zoals in het boek. Sla in een array bij `prev[i,j]` de laatste knoop op van het pad van i naar j (dus de knoop net vóór j).

Initialiseer `prev[i,j] = i` als (i, j) een tak is, en zet op 'ongedefinieerd' anders.

Pas binnen de drie loops, als aan de test voldaan is, deze waarde als volgt aan `prev[i,j] = prev[k,j]`.

We kunnen nu het pad van achter naar voren reconstrueren.

(Ik heb geen idee waarom het boek niet de array `next[,]` bijhoudt waarmee het pad vooruit gereconstrueerd kan worden.)

(ii) De oude methode, dus in oude uitwerkingen nog te lezen. Houdt een array `via[i,j]` bij dat de laatste keer is dat k op het pad werd gekozen. De waarde is dus het hoogste knoopnummer op het pad. Als $k =$ `via[i,j]` dan kunnen we recursief het pad bepalen door nu de paden van i naar k en van k naar j te bepalen.

oeps. Er is een verschil tussen afstand (een getal) en een pad (een rij knopen). Het algoritme van Floyd berekent niet één afstand, maar de afstand tussen elk paar i, j. Dus er moeten ook net zoveel paden worden opgeslagen.

5a) **(i)** $O(n * m)$, for a text of length n
(ii) $O(n)$

b)

k	1	2	3	4	5	6	7	8
$F(Link(k))$	0	1	1	1	1	2	3	4

c) nb. F(k) = FLink(k)

0	0	1	2	3	0	0	1	2	3	0	1	2	3	4
1 ≠ 0, F(1)=0														
	1≠0, F(1)=0													
		1=1	2=2	3=3	0=0	1≠0, F(5) = 1								
						1≠0, F(1) = 0								
							1=1	2=2	3=3	...				4=4