Dat102 Oblig5

???

April 22, 2025

Oppgåve 1

Vi skal lagre følgjande bilnummer med ein hash-tabell med storleik 10 og hash-funksjonen som returnerer siste siffer i bilnummeret:

- EL65431 → 1
- TA14374 → 4
- ZX87181 → 1
- EL47007 → 7
- VV50000 → 0
- UV14544 → 4
- EL32944 → 4

a) Open adressering med linear probing, steglengde 1

Vi set inn i tabellen med lineær probing dersom det oppstår kollisjon:

Index	Bilnummer
0	VV50000
1	EL65431
2	ZX87181
3	
4	TA14374
5	UV14544
6	EL32944
7	EL47007
8	
9	

Forklaring:

- EL65431 går til 1.
- ZX87181 kolliderer med 1 → neste ledige er 2.
- TA14374 går til 4.
- UV14544 kolliderer med 4 → neste ledige er 5.
- EL32944 kolliderer med 4, 5 → neste ledige er 6.

b) Kjeda lister – tabell med 10 posisjonar

Her lagrar vi kolliderande element i ei liste på kvar posisjon:

Index	Bilnummer
0	VV50000
1	EL65431, ZX87181
2	
3	
4	TA14374, UV14544, EL32944
5	
6	
7	EL47007
8	
9	

c) Gjennomsnittleg antal kall av `equals`-metoden

Open adressering:

- EL65431: 1 kall
- ZX87181: 2 kall (kolliderer med 1, så må sjekke 1 og 2)
- TA14374: 1 kall
- UV14544: 2 kall (4 og 5)
- EL32944: 3 kall (4, 5, 6)
- EL47007: 1 kall
- VV50000: 1 kall

Totalt: $1 + 2 + 1 + 2 + 3 + 1 + 1 = 11$
Gjennomsnitt: $\frac{11}{7} \approx 1.57$

Kjeda lister:

- EL65431: 1 kall

- ZX87181: 2 kall (må sjekke EL65431 først)
- TA14374: 1 kall
- UV14544: 2 kall (må sjekke TA14374 først)
- EL32944: 3 kall (må sjekke TA14374 og UV14544 først)
- EL47007: 1 kall
- VV50000: 1 kall

Totalt: $1 + 2 + 1 + 2 + 3 + 1 + 1 = 11$
Gjennomsnitt: $\frac{11}{7} \approx 1.57$
Konklusjon: For dette datasettet og tabellstorleiken gir både open adressering med lineær probing og kjeda lister same gjennomsnittleg antal kall av `equals`-metoden.

d/e) Gjennomsnittleg antal kall av `equals`-metoden ved søk etter element som *ikkje* finst i tabellen

Vi antar at hash-verdiane til elementa vi søker etter er uniformt fordelt over dei 10 plassane i tabellen.
Open adressering med lineær probing:
Ved open adressering må vi lineært søke gjennom posisjonane frå hash-verdien og vidare, heilt til vi finn ein tom plass (indikasjon på at elementet ikkje er i tabellen). Antal kall av `equals`-metoden tilsvarar då antal utførte samanlikningar før vi møter ei tom celle.
Ved fyllingsgrad

$$\alpha = \frac{7}{10} = 0.7$$

$$\left(\frac{\text{antal element i tabellen}}{\text{antall plasser i tabellen}}\right)$$

og bruk av lineær probing er det ein kjent formel for gjennomsnittleg antal samanlikningar ved *mislykka* søk:

$$C_{\text{mislykka}} = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

Setter vi inn $\alpha = 0.7$:

$$C_{\text{mislykka}} = \frac{1}{2}\left(1 + \frac{1}{1-0.7}\right) = \frac{1}{2}\left(1 + \frac{1}{0.3}\right) = \frac{1}{2}(1 + 3.\overline{3}) = \frac{1}{2} \cdot 4.\overline{3} \approx 2.17$$

Svar: Ca. 2.17 kall av `equals`-metoden i gjennomsnitt.
Kjeda lister:
Her er gjennomsnittleg antal kall ved mislykka søk lik den gjennomsnittlege lengda på listene der ein søker, altså lik fyllingsgraden $\alpha = \frac{7}{10} = 0.7$.
Svar: Ca. 0.7 kall av `equals`-metoden i gjennomsnitt.

Konklusjon: Ved søk etter element som ikkje finst i tabellen, er kjeda lister klart meir effektivt (0.7 samanlikna med 2.17 kall i snitt), særleg når tabellen byrjar å bli full. Dette er ein av grunnane til at kjeda hashing ofte er å føretrekke ved høg lastfaktor.

f) Ny hash-funksjon

La oss definere hash-funksjonen ϕ : bilnummer $\rightarrow Z_{20}$ hvor

$$\phi(S) = \left(\sum_{\substack{c \in S \\ c \in \{0,1,\ldots,9\}}} int(c) \right) \mod 20$$

Bilnummer	**Siffer-sum**	**Hash**	**Probing**	**Plassering**
EL65431	19	19	-	19
TA14374	19	19	19 opptatt $\rightarrow$ 0 ledig	0
ZX87181	25	5	-	5
EL47007	18	18	-	18
VV50000	5	5	5 opptatt $\rightarrow$ 6 ledig	6
UV14544	18	18	18, 19, 0 opptatt $\rightarrow$ 1 ledig	1
EL32944	22	2	-	2

Oppgave 2

a) Kvifor er metoden ikkje optimal?
Metoden `skrivVerdierRek` går gjennom *heile* treet, uansett om delar av treet ikkje kan innehalde verdiar i det gitte intervallet. Dette fører til unødvendige kall til `compareTo()` og rekursive funksjonar. For eksempel: om grenseverdiane er mellom 30 og 50, men venstre subtre berre inneheld verdiar under 10, er det ineffektivt å søke der.
b) Forbedra metode med færre kall til `compareTo()`
Ved å sjekke om det i det heile tatt er mogleg at verdiar i eit subtre ligg innanfor intervallet, kan vi unngå unødvendige kall:

```
private void skrivVerdierRek(BinaerTreNode<T> t, T min, T maks) {
    if (t != null) {
        // Gå til venstre kun om det er mogleg å finne verdiar >= min
        if (t.getElement().compareTo(min) > 0) {
            skrivVerdierRek(t.getVenstre(), min, maks);
        }

        // Skriv ut element dersom det ligg innanfor intervallet
        if (t.getElement().compareTo(min) >= 0 &&
            t.getElement().compareTo(maks) <= 0) {
            System.out.print(t.getElement() + " ");
        }

        // Gå til høgre kun om det er mogleg å finne verdiar <= maks
        if (t.getElement().compareTo(maks) < 0) {
            skrivVerdierRek(t.getHogre(), min, maks);
        }
    }
}
```

Denne versjonen unngår unødvendige søk og reduserer antall kall til `compareTo()` vesentleg ved å utnytte strukturen i det binære søketreet.

Oppgåve 3 – Balansering av binært søketre

a) Endringar i `BinaerTreNode` for å lagre høgde

Vi legg til ein ny objektvariabel `hogdeU` og tilhøyrande `get`- og `set`-metodar. Vi modifiserer også konstruktøren:

```
public class BinaerTreNode<T> {
    private T element;
    private BinaerTreNode<T> venstre, hogre;
    private int hogdeU;

    public BinaerTreNode(T element) {
        this.element = element;
        this.venstre = null;
        this.hogre = null;
        this.hogdeU = 1; // Eit enkelt tre har høgde 1
    }

    public int getHogdeU() {
        return hogdeU;
    }

    public void setHogdeU(int hogdeU) {
        this.hogdeU = hogdeU;
    }

    // Andre get-/set-metodar for venstre, hogre, og element...
}
```

b) Sjekk om treet er balansert

Eit binært tre er balansert dersom forskjellen i høgde mellom venstre og høgre undertre for kvar node er maksimalt 1. Vi antar at `hogdeU` er korrekt oppdatert i alle nodar:

```
public boolean erBalansert(BinaerTreNode<T> node) {
    if (node == null) return true;

    int venstreH = (node.getVenstre() == null) ? 0 : node.getVenstre().getHogdeU();
    int hogreH   = (node.getHogre() == null) ? 0 : node.getHogre().getHogdeU();

    if (Math.abs(venstreH - hogreH) > 1) {
        return false;
    }

    return erBalansert(node.getVenstre()) && erBalansert(node.getHogre());
}
```

c) (Frivillig) Oppdatere `hogdeU` i `leggTil`-metoden

Ved innsetting kan vi rekursivt oppdatere `hogdeU` slik at kvar node på vegen får oppdatert sin lokale høgde basert på barnegreinene:

```
private BinaerTreNode<T> leggTil(BinaerTreNode<T> node, T verdi) {
    if (node == null) {
        return new BinaerTreNode<>(verdi);
    }

    if (verdi.compareTo(node.getElement()) < 0) {
        node.setVenstre(leggTil(node.getVenstre(), verdi));
    } else {
        node.setHogre(leggTil(node.getHogre(), verdi));
    }

    int venstreH = (node.getVenstre() == null) ? 0 : node.getVenstre().getHogdeU();
    int hogreH   = (node.getHogre() == null) ? 0 : node.getHogre().getHogdeU();

    node.setHogdeU(1 + Math.max(venstreH, hogreH));
    return node;
}
```

Oppgave 4 - Hauger

a)

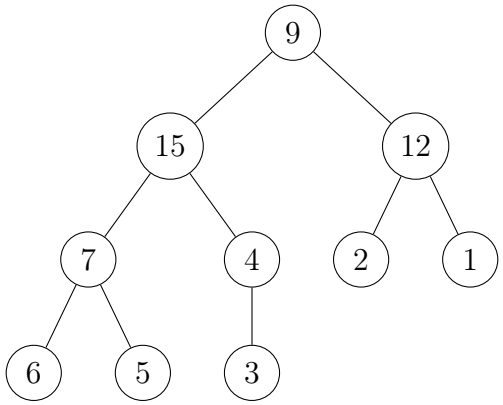
En haug er en spesiell type binærtre som tilfredsstiller to egenskaper:

1. Formegenskapen: Treet er komplett, dvs. alle nivåer er fyllt untatt (muligens) i det siste nivået av treet hvor vi legger til noder fra venstre til høyre.

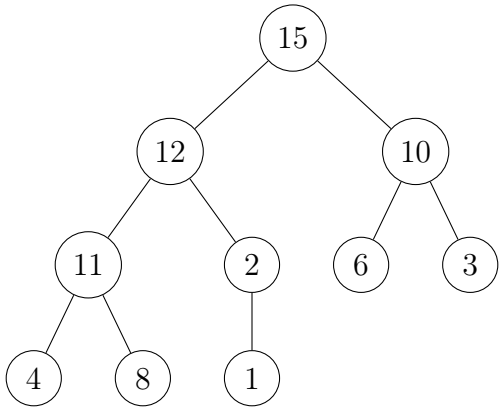
2. For en makshaug er hvert element mindre enn eller lik foreldren. For en minhaug er hvert element større enn eller like foreldren.

b)

a[0]

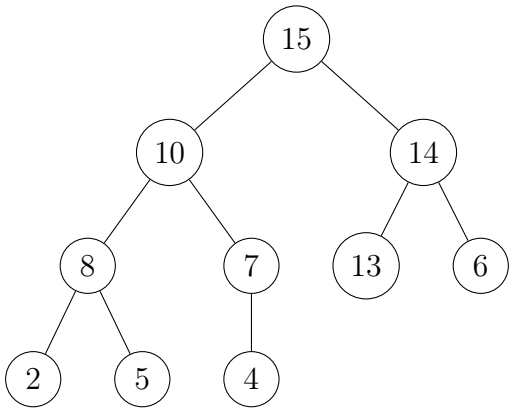


Vi ser at første tabellen ikke opfyller kravet til an makshaug siden $9 < 15$.
b[0]



Dette er tydelig en makshaug siden hver foreldrenode er større enn eller lik begge barn.

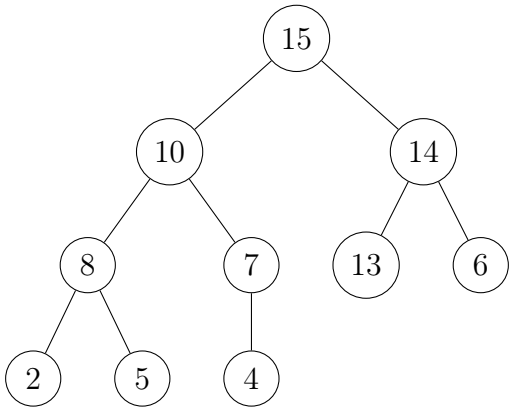
c[0]



Dette er også tydelig en makshaug.

c)

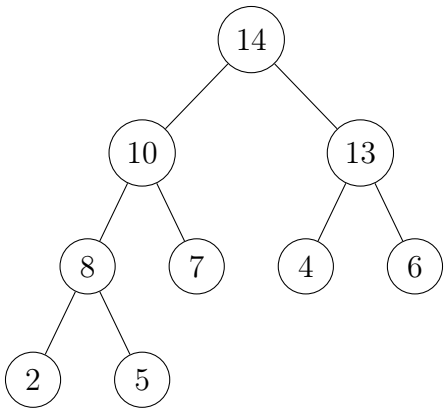
d[0] (Start)



i) Fjerne 15 og reorganisere til makshaug:

1. Fjern 15, erstatt med 4 (siste element).
2. Sammenlign 4 med barn 10 og 14. Størst av barna er 14 → bytt.
3. Nå står 4 under 14, sammenlign med barn 13 og 6. Bytt med 13.

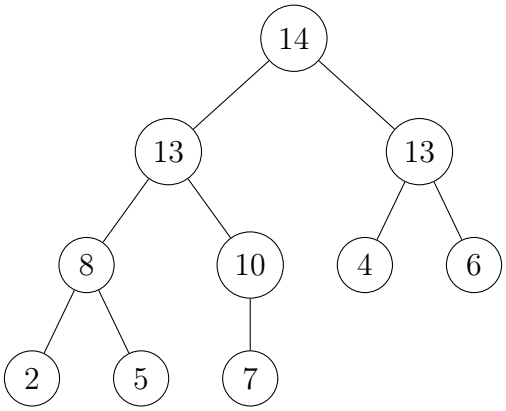
Resultat-tre etter reorganisering:



ii) Sett inn nytt element 13 og reorganiser:

1. Legg 13 inn bakerst.
2. Forelderen til 13 er 7. $13 > 7 \Rightarrow$ bytt.
3. 13 er nå barn av 10, og $13 > 10 \Rightarrow$ bytt.

Resultat-tre:



d) Hvordan brukes dette i sortering (Heapsort)?

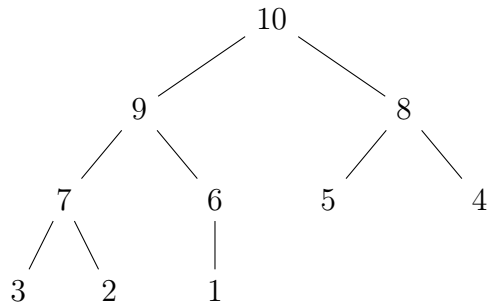
Vi bygger først en makshaug av elementene. Deretter:

1. Fjerner roten (maksverdi) og legger bakerst i tabellen.
2. Flytter siste element til roten og reorganiserer haugen.
3. Gjentas til alle elementer er fjernet.

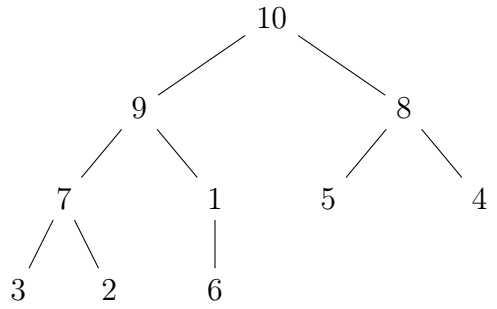
Resultatet blir sortert tabell i stigende rekkefølge.

e) Lage en minimumshaug fra: `[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]`

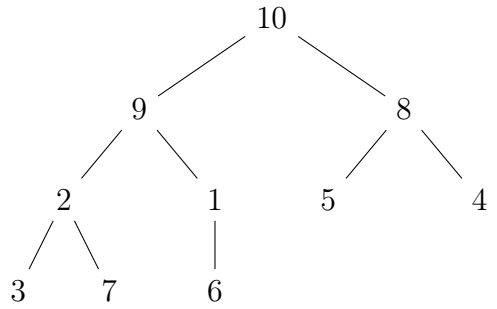
Begynn med treet (indeksert fra 1):



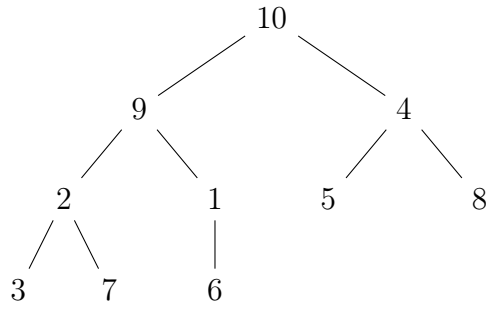
Start reparerNed fra siste interne node (indeks $\lfloor n/2 \rfloor - 1 = 4$), og gå mot roten. Ved hver rotasjon, flytt ned noden om den er større enn et barn.
reparerNed(4) Node 6 har barn 1. $1 < 6$, så vi bytter:



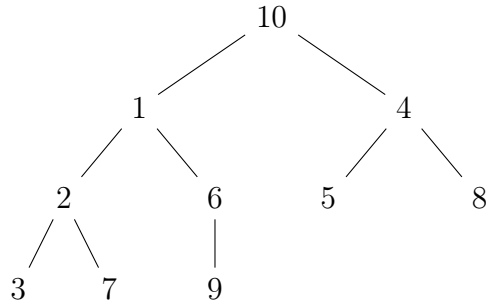
reparerNed(3) Node 7 har barn 3 og 2. Minst er 2, så bytt:



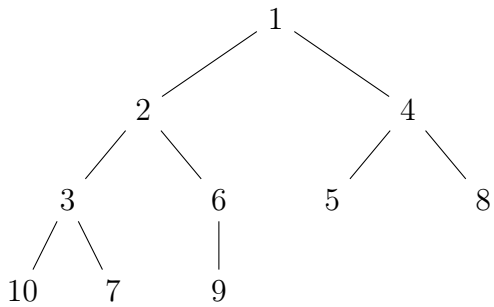
reparerNed(2) Node 8 har barn 5 og 4. Minst er 4, så bytt:



reparerNed(1) Node 9 har barn 2 og 1. Minst er 1, så bytt. Deretter har vi $6 < 9$, så bytt igjen



reparerNed(0) Node 10 har barn 1 og 4. Minst er 1, så bytt. Deretter har 10 barn 2 og 9, vi bytter med 2. Deretter har 10 barn 3 og 7 og vi bytter med 3.



Ferdig minimumshaug. Array nå: `[1, 2, 4, 3, 6, 5, 8, 10, 9]`

Oppgave 5

Vi ønsker å finne de k minste elementene i en usortert tabell med n elementer. Å sortere hele tabellen gir korrekt resultat, men kan være unødvendig når $k \ll n$. Her modifiserer vi tre sorteringsmetoder for å løse problemet mer effektivt.

1. Insertion Sort (modifisert)

- Sorter de første k elementene med vanlig insertion sort.
- For hvert av de resterende $n - k$ elementene:
  - Hvis elementet er mindre enn det største av de k sorterte, sett det inn på riktig plass i den sorterte listen og kast det største.
- Dette opprettholder en sortert liste over de k minste elementene.

Tidskompleksitet:

$$\mathcal{O}(k^2 + (n-k) \cdot k) = \mathcal{O}(nk)$$

(for små k er dette mye bedre enn full sortering).

2. Selection Sort (modifisert)

- Utfør kun de første k iterasjonene av selection sort:
  - I hver iterasjon, finn det minste gjenværende elementet og flytt det til fronten.

Tidskompleksitet:

$$\mathcal{O}(n \cdot k)$$

(for små k er dette effektivt; vi unngår full sortering).

3. Heap Sort (modifisert)

- Bygg en **maks-haug** med de første k elementene.
- For hvert av de resterende $n - k$ elementene:
  - Hvis elementet er mindre enn roten i haugen (det største), erstatt roten og `reparer ned` for å opprettholde haug-egenskapen.
- Når alle elementer er behandlet, inneholder haugen de k minste elementene (ikke nødvendigvis sortert).
- For å få dem sortert, bruk `heap sort` på den lille haugen.

Tidskompleksitet:

$$\mathcal{O}(k + (n-k) \cdot \log k + k \cdot \log k) = \mathcal{O}(n \log k)$$

Oppgave 6

a)

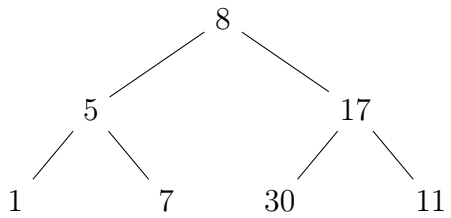
Viss vi lar τ representere et binærtree så har vi at det er balansert viss

$$\forall v \in V(\tau) : |hoyde(v.venstre) - hoyde(v.hoyre)| \leq 1$$

Med andre ord, for hver node i treet må høydeforskjellen mellom venstre og høyre undertre være mindre enn eller lik 1.

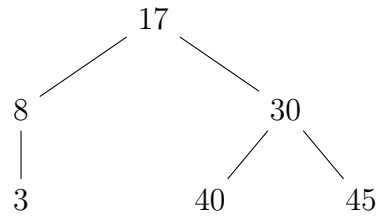
b)

Rotnoden har venstre undertre med 2 mer dybde enn høyre undertre. Når vi gjør en høyrerotasjon om rotnoden får vi dette treet:



c)

Treet er ubalansert fordi node 30 har venstre undertree med dybde 0 og høyre undertree med dybde 2. Vi roterer 30 til venstre for å opnå:



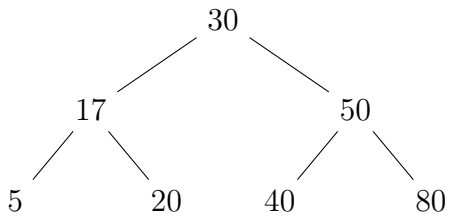
d)

Hvis et BS-tre er balansert er det $\mathcal{O}(\log n)$ for instetting, sletting og søk. Viss vi lar treet degenerere går det imot å være en lenket liste som er $\mathcal{O}(n)$ for de samme operasjonene.

Oppgave 7

a)

Vi setter inn $[20, 50, 30, 5, 40, 80, 17]$ i et 2-3 tre i den gitte rekkefølgen. Dette resulterer i følgene tre:



b)

1. Ingen noder har nøyaktig ett barn: **sant**
2. Treet inneholder maksimalt en 3-node: **usant**
3. Alle blad er på samme nivå: **sant**
4. Dersom treet bare inneholder 2-noder, så vil det tilsvare et balansert BS-tre: **usant**
5. Alle 3-noder må være blad: **usant**

Oppgave 8

a)

Viss vi antar at vi velger noder med lavest numerisk verdi først blir det:

1. 9
2. 6
3. 3
4. 1
5. 2
6. 4
7. 5
8. 7
9. 8

Her starter vi i roten og velger den laveste verdi noden som ikke er besøkt. Vi legger den laveste verdi ubesøkte noden på en stack (uten duplikat) og besøker den.

b)

Viss vi igjen antar at vi prioriterer tilgjengelige noder med lavest numerisk verdi blir det:

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9

Viss vi heller antar at vi prioriterer tilgjengelige noder med høyest numerisk verdi blir det:

1. 1
2. 4
3. 3
4. 2
5. 6
6. 5
7. 8
8. 9
9. 7

c)

Vi ønsker å finne en topologisk ordning av nodene i grafen under ved hjelp av algoritmen fra boka/forelesningen.

Grafen inneholder følgende kanter:

$$1 \to 2, \quad 1 \to 3, \quad 2 \to 4, \quad 3 \to 2, \quad 3 \to 4, \quad 4 \to 5, \quad 6 \to 5, \quad 6 \to 7, \quad 7 \to 5$$

For å finne en topologisk ordning, kan vi bruke Kahn's algoritme (BFS-metoden), som er en iterativ metode som fjerner noder med in-degree lik 0 og reduserer in-degrees for tilknyttede noder. Prosessen kan deles opp i følgende trinn:

1. Beregn in-degree for hver node.
2. Start med de nodene som har in-degree lik 0 (dvs. noder uten innkommende kanter).
3. Legg nodene med in-degree lik 0 i en kø.
4. Mens køen ikke er tom:
   - Fjern noden fra køen og legg den til i den topologiske ordningen.
   - Reduser in-degree for alle naboene til noden som er fjernet.
   - Hvis en nabonode får in-degree lik 0, legg den til i køen.
5. Hvis alle nodene er behandlet, har vi funnet en topologisk ordning. Hvis det er noen noder igjen med in-degree større enn 0, betyr det at grafen inneholder en syklus, og en topologisk ordning er ikke mulig.

In-degree for hver node:

$$\text{In-degree}(1) = 0, \quad \text{In-degree}(2) = 2, \quad \text{In-degree}(3) = 1, \quad \text{In-degree}(4) = 2,$$

$$\text{In-degree}(5) = 3, \quad \text{In-degree}(6) = 0, \quad \text{In-degree}(7) = 1$$

Start med noder med in-degree 0: $1, 6$
Kø: $\{1, 6\}$
Behandling:

- Fjern node 1, legg den til i topologisk ordning.
- Reduser in-degree for 2 og 3.
- Nå har vi 2 og 3 i køen.
- Fjern node 3, legg den til i topologisk ordning.
- Reduser in-degree for 2 og 4.
- Nå har vi 2 og 4 i køen.
- Fjern node 2, legg den til i topologisk ordning.
- Reduser in-degree for 4.
- Nå har vi 4 i køen.
- Fjern node 4, legg den til i topologisk ordning.
- Reduser in-degree for 5.
- Nå har vi 6 og 7 i køen.
- Fjern node 6, legg den til i topologisk ordning.

- Reduser in-degree for 5 og 7.
- Nå har vi 7 i køen.
- Fjern node 7, legg den til i topologisk ordning.
- Reduser in-degree for 5.
- Nå har vi 5 i køen.
- Fjern node 5, legg den til i topologisk ordning.

Topologisk ordning: $1, 3, 2, 4, 6, 7, 5$

d)

For å modifisere algoritmen slik at den gir en melding om hvorvidt en topologisk ordning finnes eller ikke, kan vi endre prosessen som følger:

1. Beregn in-degree for hver node.
2. Start med de nodene som har in-degree lik 0 (dvs. noder uten innkommende kanter).
3. Legg nodene med in-degree lik 0 i en kø.
4. Mens køen ikke er tom:
    - Fjern noden fra køen og legg den til i den topologiske ordningen.
    - Reduser in-degree for alle naboene til noden som er fjernet.
    - Hvis en nabonode får in-degree lik 0, legg den til i køen.
5. Hvis alle nodene er behandlet og ingen noder er igjen med in-degree større enn 0, har vi funnet en topologisk ordning.
6. Hvis det er noen noder igjen med in-degree større enn 0, betyr det at grafen inneholder en syklus, og en topologisk ordning er ikke mulig. Da skal algoritmen gi en melding om at det ikke finnes en topologisk ordning.

e)

Ved hver node ser vi på avstanden til alle tilgjengelige noder som ikke er besøkte og velger den med minst vekt og noterer at den er besøkt. Ved bruk av denne algoritmen har vi at den minimale avstanden fra A til H er 7 ($A \to B \to C \to H$)