# Eksamen i IN2010 høsten 2022

1. Desember 2022

## Om eksamen

- Eksamen består av en bitteliten oppvarming, etterfulgt av to hoveddeler.
- Den første delen består av små oppgaver, som rettes automatisk. Det gjøres ingen forskjell mellom ubesvart og feil svar; det betyr at det lønner seg å svare på alle oppgavene.
- Den andre delen består av litt større oppgaver hvor du i større grad må programmere (pseudokode), skrive og resonnere.
- Alle besvarelser skal skrives inn i Inspera og det er ingen mulighet for opplasting av håndskrevede svar.
- Ingen hjelpemidler er tillatt.

## Kommentarer og tips

- Det er lurt å lese raskt gjennom eksamen før du setter i gang. Hele oppgavesettet er lagt ved som PDF.
- Det kanskje viktigste tipset er *å lese oppgaveteksten svært nøye*.
- Pass på at du svarer på nøyaktig det oppgaven spør om.
- Pass på at det du leverer fra deg er klart, presist og enkelt å forstå, både når det gjelder form og innhold.
- Hvis du står fast på en oppgave, bør du gå videre til en annen oppgave først.
- Alle implementasjonsoppgaver skal besvares med *pseudokode*. Det viktige er at pseudokoden er *lett forståelig, entydig og presis*.
- En *lett forståelig, entydig og presis* forklaring med naturlig språk, kan være mer poenggivende enn pseudokode som er vanskelig å forstå, tvetydig eller upresis.

## Oppvarming 2 poeng

(a) Hva er en algoritme? Svar kort (maks fire setninger).

(b) Hva er en datastruktur? Svar kort (maks fire setninger).

Vi er ikke ute etter et «fasitsvar». Vi er ute etter å høre din forståelse av begrepene, og alle rimelige svar gir full uttelling.

## Litt av hvert 10 poeng

(a) Binære trær består av noder som har opptil to barn.

(b) Hvis en algoritme bruker $\mathcal{O}(n)$ antall steg sier vi den har lineær kjøretidskompleksitet.

(c) Kjøretidskompleksitet til Mergesort er $\mathcal{O}(n \cdot \log(n))$.

(d) Det er umulig å sjekke om et array er sortert i $\mathcal{O}(n)$.

(e) Binærsøk på arrayer er betydelig raskere enn binærsøk på lenkede lister.

(f) Kjøretidskompleksitet sier noe om hvor mange steg en algoritme bruker i forhold til størrelsen på input.

(g) Dersom en algoritme har bedre kjøretidskompleksitet enn en annen algoritme, så vil den alltid bruke færre steg uansett størrelse på input.

(h) Huffman-koding brukes for å komprimere data.

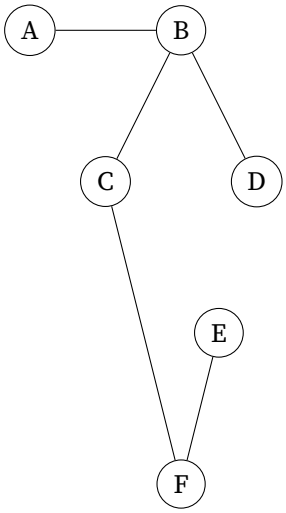
(i) Rotnoden av et tre er den eneste noden som ikke har en forelder.

(j) En graf med $n$ noder kan ikke har mer enn $n$ kanter.

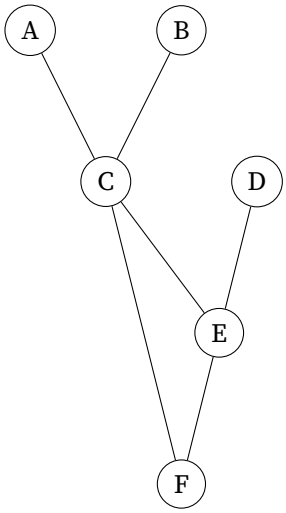
## Grafegenskaper 10 poeng

Her er to grafer:

$G_1$

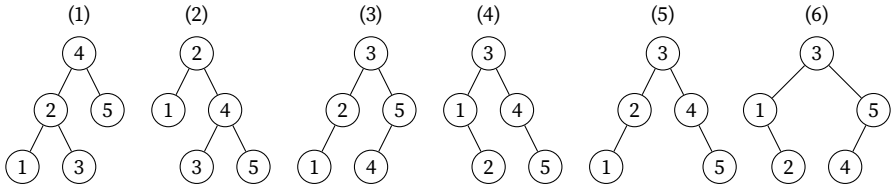



$G_2$



## Balanserte søketrær 6 poeng

Følgende er alle AVL-trær som inneholder tallene 1 til 5.




(a) Hvilket av AVL-trærne ovenfor får man hvis man legger inn tallene 1 til 5 i rekkefølgen $2, 4, 3, 1, 5$? Oppgi svaret som et tall mellom 1 og 6.

(b) Hvor mange av trærne ovenfor kan fargelegges som et rød-svart tre?

(c) Hvis vi legger til $6$ i alle AVL-trærne ovenfor, i hvilke trær vil det forekomme rotasjoner? Oppgi svaret som summen av alle trærne det gjelder. Altså, hvis det kun forekommer rotasjoner i det første treet, er svaret $1$, og hvis det forekommer rotasjoner i alle trærne, er svaret $21$ (fordi $1 + 2 + 3 + 4 + 5 + 6 = 21$).

## Litt om prioritetskøer og binære heaps 10 poeng

Med *heap* mener vi arrayer som representerer binære min-heaps.

(a) En array med $n$ elementer kan gjøres om til en heap i $\mathcal{O}(n)$.

(b) En min-heap blir en max-heap ved å reversere arrayet.

(c) Alle elementer på dybde $d$ i en heap er mindre enn alle elementer på dybde $d+1$.

(d) Innsetting i en heap med $n$ elementer er $\mathcal{O}(\log(n))$ i verste tilfelle.

(e) Et AVL-tre kan brukes som en prioritetskø med samme kjøretidskompleksitet som en heap.

(f) Nodene langs en sti fra rotnoden til en løvnode i en heap er ordnet fra minst til størst.

(g) Dersom vi bytter plass på to søskennoder i en heap, er det fremdeles en heap.

(h) En prioritetskø med konstant tid på alle operasjoner ville gjort at Dijkstra hadde kjøretidskompleksitet $\mathcal{O}(\|V|+|E|)$.

(i) Man kan finne det *største* elementet i en *min*-heap i $\mathcal{O}(\log(n))$.

(j) I en heap ligger *over halvparten* av elementene på de *to dypeste* nivåene.

## Kjøretid på grafalgoritmer 8 poeng

For hver grafalgoritme, kryss av på den (laveste) korrekte kjøretidskompleksiteten.

| | $O(1)$ | $O(\|V\| + \|E\|)$ | $O((\|V\| + \|E\|) \cdot \log(\|V\|))$ | $O(\|V\| \cdot \|E\|)$ |
|---|---|---|---|---|
| `DFSFull` | | | | |
| `Prim` | | | | |
| `TopSort` | | | | |
| `BellmanFord` | | | | |

*Korte* beskrivelser av algoritmene:

- `DFSFull`: Besøker alle noder i en graf nøyaktig én gang (dybde-først)
- `Prim`: Finner et minimalt spenntre av en gitt graf
- `TopSort`: Gir en topologisk ordning av nodene i en gitt graf
- `BellmanFord`: Finner korteste stier fra én til alle andre noder

## Linear probing 10 poeng

(a) Vi starter med et tomt array på størrelse $10$.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Hashfunksjonen du skal bruke er $h(k, N) = k \mod N$, som for dette eksempelet blir det samme som $h(k, 10) = k \mod 10$. Altså hasher et tall til sitt siste siffer.

Bruk *linear probing* til å sette inn disse tallene i den gitte rekkefølgen:

$$21, 54, 82, 10, 20, 44$$

Fyll ut tabellen slik den ser ut etter alle tallene er satt inn med linear probing. Skriv svaret som en kommaseparert liste, der _ kan brukes for å indikere en tom plass.

(b) Forklar kort hvordan algoritmen for innsetting ved linear probing fungerer, og skisser algoritmen med pseudokode. Du kan anta at arrayet i input ikke er fullt, og at du har en hashfunksjon $h$ slik at $h(k, N)$ gir et tall mellom $0$ og $N - 1$ for en vilkårlig nøkkel $k$. Ledig plass i arrayet er indikert ved **`null`**.

**Input:** Et array A av størrelse $N$, en nøkkel $k$ og verdi $v$
**Output:** Et array som inneholder $(k, v)$

```
1 Procedure LinearProbingInsert(A, k, v)
  | // ...
```

## Garbage collection 8 poeng

Mange moderne programmeringsspråk har en *garbage collector*, som er en prosedyre som frigjør minne som garantert ikke vil brukes i programmet lenger. Du skal utlede en enkel algoritme for garbage collection.

Vi kan anta at alt som lagres er *objekter*, der et objekt kan referere til andre objekter. Vi lar en $G = (V, E)$ være en objektgraf, der $V$ representerer alle objektene som er opprettet, og en (rettet) kant fra $u$ til $v$ betyr at objektet $u$ har en referanse til objektet $v$. I tillegg har vi en mengde $R$ med alle objektene som kan refereres til direkte (typisk objekter som refereres til av programvariabler). Alle objekter i $R$ er også i $V$. Objekter det *ikke* finnes en referanse til via objekter i $R$ er garantert å ikke bli brukt i programmet, og skal derfor frigjøres.

Det betyr at ingen av objektene i $R$ skal frigjøres, og heller ingen objekter som kan nås gjennom referanser fra objekter i $R$ skal frigjøres. Objektene som skal frigjøres er ikke i $R$, og kan heller ikke nås fra noe objekt i $R$.

Anta at du har en prosedyre `Free` som frigjør et objekt. Du skal gi en prosedyre `GarbageCollect` som tar en graf $G = (V, E)$ og en mengde $R$ som input, og kaller `Free` på alle objekter det *ikke* kan nås fra $R$.

---

**Input:** En objektgraf $G = (V, E)$ og en mengde $R$ med objekter
**Output:** Frigjør alle objekter det ikke finnes en referanse til

```
1 Procedure GarbageCollect(G, R)
  | // ···
```

---

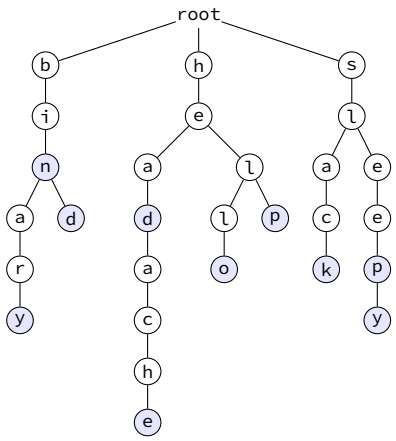
# Auto complete 6 poeng

De fleste teksteditorer har funksjonalitet for «auto completion», altså der det dukker opp en liten boks med mulige måter å fullføre et påbegynt ord. Nedenfor gir vi to mulige datastrukturer som kan brukes for å implementere en enkel mekanisme for å fullføre et påbegynt ord.

Vi viser eksempler på hvordan datastrukturene ser ut dersom de lagrer ordene: `bin`, `binary`, `bind`, `head`, `headache`, `hello`, `help`, `slack`, `sleep`, `sleepy`.

## Strategi 1

Den første strategien er å bruke et *prefiks-tre* som holder på strenger. Et prefiks-tre har en rot. Hvis en streng $s$ er lagt inn i treet, har rotnoden et barn med $s$ sin første bokstav. Den noden har igjen en peker til $s$ sin andre bokstav, og så videre. Den siste bokstaven i $s$ er markert som en *terminal* node (terminalnodene er fargelagt blå i eksempelet). Det vil si at hvis du følger en sti fra rotnoden til en terminalnode, så har du stavet et ord som er lagt inn i treet.




Fra et prefiks-tre er det enkelt å finne alle måter å fullføre et påbegynt ord. Det gjøres ved å returnere alle stier fra rotnoden som begynner med bokstavene fra et gitt ord og ender i en terminalnode.

## Strategi 2

Den andre strategien er å bruke et hashmap, der vi lar hver prefiks av et ord mappe til en liste av alle mulige måter å fullføre ordet. Hvis `H` er et hashmap, og $s$ er en streng som skal legges til, ser vi på hver prefiks $p$ av $s$ og legger $s$ til i listen `H`$[p]$. I eksempelet ved siden av ser dere hvordan hashmapet kan se ut dersom alle eksempelordene er lagt (merk at noen av listene er kortet ned, indikert ved «...»).

| Nøkkel | Verdi |
|---|---|
| `""` | `"bin", ..., "sleepy"` |
| `"b"` | `"bin", "binary", "bind"` |
| `"bi"` | `"bin", "binary", "bind"` |
| `"bin"` | `"bin", "binary", "bind"` |
| `"bina"` | `"binary"` |
| `"binar"` | `"binary"` |
| `"binary"` | `"binary"` |
| `"bind"` | `"bind"` |
| `"h"` | `"head", ... "help"` |
| `"he"` | `"head", ... "help"` |
| `"hea"` | `"head", "headache"` |
| `"head"` | `"head", "headache"` |
| `"headа"` | `"headache"` |
| `"headac"` | `"headache"` |
| `"headach"` | `"headache"` |
| `"headache"` | `"headache"` |
| `"hel"` | `"hello", "help"` |
| `"hell"` | `"hello"` |
| `"hello"` | `"hello"` |
| `"help"` | `"help"` |
| `"s"` | `"slack", "sleep", "sleepy"` |
| `"sl"` | `"slack", "sleep", "sleepy"` |
| `"sla"` | `"slack"` |
| `"slac"` | `"slack"` |
| `"slack"` | `"slack"` |
| `"sle"` | `"sleep", "sleepy"` |
| `"slee"` | `"sleep", "sleepy"` |
| `"sleep"` | `"sleep", "sleepy"` |
| `"sleepy"` | `"sleepy"` |

Fra et slikt hashmap er det svært enkelt å finne alle måter å fullføre et påbegynt ord. Det gjøres ved å slå opp det påbegynte ordet i hashmapet og returnere listen.

*Drøft fordeler og ulemper ved de to ulike strategiene både med tanke på kjøretid og minnebruk*

## Bucket queue 10 poeng

En *bucket queue* (eller en bøttekø) er en datastruktur som er inspirert av *bucket sort* og kan brukes som en *prioritetskø*.

I en bucket queue kan elementer bli lagt til med en prioritet mellom $0$ og $N-1$, der $N$ er et positivt heltall som fastsettes når køen opprettes. Ved et kall på `RemoveMin`, så vil et element med lavest mulig prioritet fjernes og returneres; dersom det er flere elementer med samme prioritet spiller det ikke noen rolle hvilket element som fjernes og returneres.

(a) Forklar *kort* hvilken datastruktur som passer godt for en bucket queue.

(b) Gi en algoritme for `Insert` for en bucket queue.

```
Input: En bucket queue Q med prioriteter fra 0 til N − 1, og et element x med prioritet p
Output: Q med x satt inn med prioritet p
1 Procedure Insert(Q, x, p)
  | // ···
```

(c) Gi en algoritme for `RemoveMin` for en bucket queue. Du kan anta at køen ikke er tom.

```
Input: En bucket queue Q med prioriteter fra 0 til N − 1
Output: En x med lavest mulig prioritet, som er fjernet fra Q
1 Procedure RemoveMin(Q)
  | // ···
```

(d) Anta at $N = 100$. Oppgi kjøretidskompleksiteten på `Insert` og `RemoveMin` for en kø med $n$ elementer.

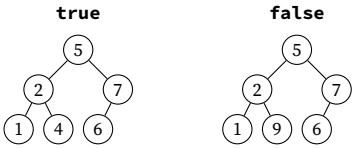
## Er binærtreet et søketre? 10 poeng

Anta at du er gitt et binærtre $B$ med unike heltall. Hvis $v$ er en node i det binære treet, så gir

- $v$.`element` heltallet som er lagret i noden
- $v$.`left` venstre barn av $v$
- $v$.`right` høyre barn av $v$

Vi ønsker å sjekke om det binære treet også er et binært *søketre*. Under finner du spesifikasjonen for algoritmen, og to eksempler på trær som henholdsvis bør gi **`true`** og **`false`**.

```
Input: Rotnoden v av et binærtre B
Output: Returnerer true hvis binærtreet er et binært
        søketre, false ellers
1 Procedure CheckBST(v)
  | // ···
```




Det finnes flere gode løsninger på dette problemet. Følgende kan være behjelpelig når du tenker på en løsning:

- Du kan anta at du har prosedyrer `FindMin` og `FindMax` som henholdsvis finner minste og største tall i det binære treet. Siden treet ikke er garantert å være et binært søketre (eller balansert) så vil disse prosedyrene ha lineær tid.
- Det kan være lurt å dele algoritmen opp ved å lage en hjelpeprosedyre.

(a) Skriv ned egenskapen et binærtre må ha for å kunne kalles et binært *søketre*.

(b) Fullfør prosedyren over. Lavere kjøretidskompleksitet er mer poenggivende.

(c) Oppgi kjøretidskompleksiteten på algoritmen din med hensyn til antall noder i binærtreet.

## Minimal forside 10 poeng

Du er gitt en graf $G = (V, E)$ som representerer et domene, der hver node representerer en side under domenet. Dersom det finnes en kant fra $u$ til $v$, betyr det at det er en link på side $u$ som linker til $v$.

Du har fått i oppdrag å lage en forside (som blir en ny node i grafen), som skal oppfylle følgende kriterer:

- Alle sider skal kunne nås fra forsiden.
- Forsiden skal inneholde så få linker som mulig.

Du skal beskrive algoritmer som beregner hvor mange linker den nye forsiden må inneholde avhengig av om grafen er:

(a) En urettet graf. Gi en kort beskrivelse av algoritmen med naturlig språk (du kan referere til algoritmer i kurset).

(b) En rettet og asyklisk graf. Gi en kort beskrivelse av algoritmen med naturlig språk (du kan referere til algoritmer i kurset).

(c) En rettet graf (som kan inneholde sykler). Gi en kort beskrivelse av algoritmen med naturlig språk (du kan referere til algoritmer i kurset).