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.

Sensorveiledning

Sensorveiledningen inneholder et løsningsforslag på alle oppgaver. I tillegg gis det veiledning på hvordan poeng kan fordeles, men sensorene står fritt til å gi eller trekke poeng etter eget skjønn. Det legges alltid vekt på at besvarelsene er presise og enkle å forstå, både med hensyn til form og innhold.

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.

Her finnes det utrolig mange riktige svar! Vi er ikke ute etter en formell definisjon (selv om det selvfølgelig vil være poenggivende), men vil bare få studentene i gang med å tenke på algoritmer og datastrukturer. Her er mitt svar (som på ingen måte bør være førende for hvordan vi tildeler poeng):

- Algoritmer er idéene bak de programmene vi skriver. De angir hva som må gjøres for å løse et gitt problem på en presis og entydig måte. Følger du en algoritme for å løse et problem så skal du også få riktig svar til slutt, og helst på rimelig tid.
- Datastrukturer er måter å organisere data på. Stort sett vil vi organisere dataene etter hva vi oftest ønsker å gripe tak i, slik at vi kan få mer effektive algoritmer.

Vi gir full uttelling for alle rimelige svar. For å *ikke* få full uttelling må man enten la oppgaven stå ubesvart, eller skrive noe som er direkte feil. Det kan trekkes poeng dersom deler av svaret inneholder vesentlige feil.

Litt av hvert 10 poeng

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

Sant.

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

Sant.

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

Sant.

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

Usant.

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

Sant.

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

Sant.

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

Usant.

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

Sant.

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

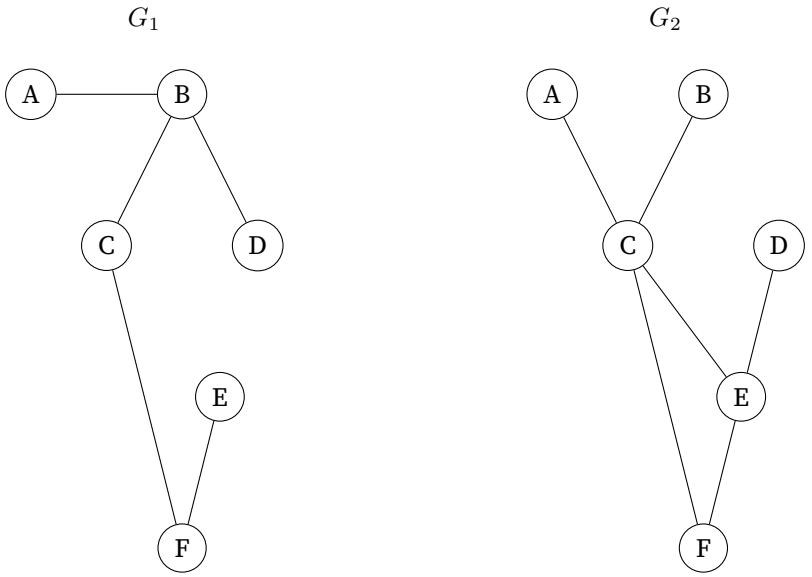
Sant.

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

Usant.

Grafegenskaper 10 poeng

Her er to grafer:



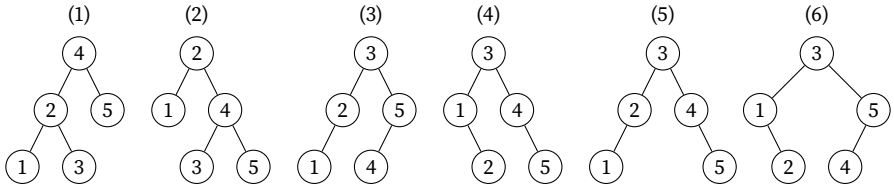
	Ingen	G_1	G_2	Begge
Grafen er sammenhengende				×
Grafen er 2-sammenhengende	×			
Grafen er et tre		×		
Grafen har mer enn to seprasjonsnoder		×		
Grafen har to (forskjellige) stier fra B til F			×	
Grafen har to komponenter	×			
Grafen er rettet	×			
Grafen inneholder en sykel			×	
Grafen blir ett tre hvis vi fjerner én kant			×	
Grafen er en enkel graf				×

Det gis 1 poeng per riktig svar.

Spørsmålet som er uthevet i rødt er uklart, og tas ut av vurderingen.

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.

5.

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

6. Alle AVL-trær 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 $1 + 2 + 3 + 4 + 5 + 6 = 21$).

$2 + 4 + 5 = 11$.

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 $\mathcal{O}(n)$.

Sant.

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

Usant.

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

Usant.

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

Sant.

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

Sant.

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

Sant.

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

Usant.

Oppgaven er litt tvetydig om hvorvidt man skal ta med seg subtrærne eller ikke. Tanken var å kun bytte om elementene i nodene. Selv hvis man tar med seg subtrærne har man ingen garanti for at det resulterende treet er *komplett*, og dermed er det heller ikke en heap. Siden begge tolkninger gir opphav til samme svar lar vi oppgaven stå.

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

Sant.

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

Usant.

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

Sant.

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

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

Det gis 2 poeng per riktig svar.

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)
  |  // ...
```

(a) Det resulterende arrayet ser slik ut:

10	21	82	20	54	44				
0	1	2	3	4	5	6	7	8	9

2 poeng for riktig svar, ingen poeng for feil. Vi trekker ikke for unøyaktigheter rundt antall tomme plasser.

(b) Linear probing fungerer ved å prøve å plassere nøkkel/verdi-paret på posisjonen som svarer til hashen av nøkkelen ($h(k, N)$). Dersom posisjonen er ledig kan vi sette inn nøkkel/verdi-paret og algoritmen er ferdig.

Dersom posisjonen er opptatt må vi:

- dersom nøkkelen på posisjonen er lik k må vi overskrive verdien;
- dersom nøkkelen på posisjonen er ulik må vi prøve neste posisjon.

```
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)
2   i ← h(k, N)
3
4   while A[i] ≠ null do
5     if key at A[i] is k then
6       break                          // Overskriv med ny verdi
7     i ← i + 1 mod N
8
9   A[i] ← (k, v)
```

Det viktigste er at studenten formidler en god forståelse av hvordan linear probing fungerer.

Opp til 5 poeng kan tildeles for en god forklaring; for full uttelling må også korrekt pseudokode være på plass. Et par ting å være oppmerksom på:

- Det trekkes 1 poeng for å ikke overskrive en lik nøkkel med ny verdi.
- Det trekkes 2 poeng dersom prosedyren vil aksessere indekser utenfor området $1 \ldots N - 1$.
- Det trekkes 1 poeng for feil bruk av hash-funksjonen (for eksempel $i \leftarrow h(i + 1, N)$ og lignende).

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

Oppgaven løses enklest ved å gjøre et dybde-først søk (`DFSVisit`) fra hver node i R, og deretter frigjøre alle noder i V som ikke er blitt besøkt.

```
Input: En objektgraf G = (V, E) og en mengde R med objekter
Output: Frigjør alle objekter det ikke finnes en referanse til
Procedure GarbageCollect(G, R)
    stack ← stack initialized with elements from R
    visited ← empty set

    while stack is not empty do
        u ← stack.pop()
        if u not in visited then
            add u to visited
            for (u, v) in E do
                add v to stack

    // Free all unvisited vertices
    for v in V do
        if v not in visited then
            Free(v)
```

Riktig idé gir opptil 4 poeng. For full uttelling må også prosedyren være beskrevet tilstrekkelig detaljert (helst med pseudokode), ikke være unødvendig komplisert og være uten feil. De kan godt anta at de har `DFSVisit` tilgjengelig, men må også komme tydelig frem hvorfor de kaller på den, og hva resultatet av kallet er.

Det trekkes to poeng for løsninger som har høyere kjøretidskompleksitet enn $\mathcal{O}(|V| + |E|)$.

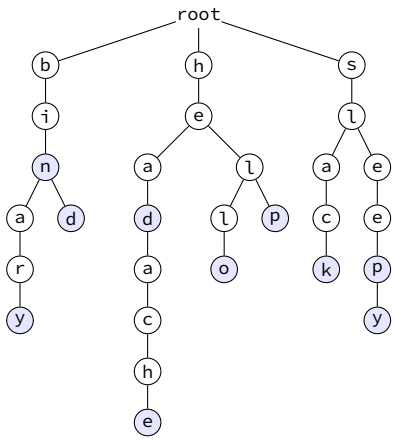
Auto complete 6 poeng

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

Dette er en oppgave som går ut på å gjøre rimelige antagelser og anvende dem til å si noe fornuftig om hvilken fremgangsmåte man bør bruke. Dette er den oppgaven fra eksamenssettet hvor sensoren er nødt til å anvende mest skjønn i poengsettingen. Tanken bak oppgaven er å få studenten til å reflektere over hvilken fremgangsmåte man bør velge i lys av kjøretidskarakteristikkene til de to strategiene. Det bør legges vekt på at teksten er godt skrevet (dette betyr ikke at det skal trekkes poeng for enkle skrivefeil, men at teksten må være tydelig og enkel å forstå; en fin tommelfingerregel er at en setning man ikke forstår etter to gjennomlesninger ignoreres). Under er et eksempel på et svar som inneholder mange refleksjoner vi ønsker å se.

Strategi 1 bruker et prefiks-tre, som er en minneeffektiv representasjon av ordlisten. Det er viktig å merke seg at denne representasjonen kan være mer plassbesparende enn å kun lagre ordlisten som en liste av strenger. Desto flere ord som deler prefiks, desto mer plassbesparende er representasjonen i forhold til en ordliste.

Å legge til et nytt ord i prefiks-treet er veldig effektivt, med tidsbruk proporsjonalt med antall bokstaver i det nye ordet.

I tillegg gir prefiks-treet opphav til en veldig naturlig algoritme for å finne måter å fullføre et påbegynt ord. Vi kan følge stien som er diktert av det påbegynte ordet, og fra subtreet til noden kan ordene samles. Dette tar lineær tid med hensyn til størrelsen av subtreet. Det vil si at dersom det på begynte ordet er den tomme strengen, så må hele ordlisten genereres, som tar lineær tid med hensyn til hele treet. Dersom vi antar at treet har høy forgrening (for eksempel 26, en for hver bokstav i det engelske alfabetet), og at størrelsen på hvert subtre er ca. like stort (som ikke er en helt rimelig antagelse), så vil hver bokstav som legges til i det påbegynte ordet korte ned antall måter å fullføre det på med 26. Det vil altså si at antall måter å fullføre ordet på minker drastisk for hver bokstav som legges til, og som gjør at det etter all sannsynlighet vil være veldig effektivt dersom vi kun lar «auto-completion»-mekanismen til å slå til dersom vi har skrevet noen få bokstaver.

Strategi 2 bruker et hashmap fra prefikser til ordlister. Dette er en veldig lite minneeffektiv representasjon. Den tomme strengen vil mappe til hele ordlisten, mens lenger ord vil mappe til kortere lister. Plassbruken vil være proporsjonalt med summen av antall bokstaver av alle ord i ordlisten.

Å legge til et nytt ord i hashmapet er veldig effektivt, med tidsbruk proporsjonalt med antall bokstaver i det nye ordet. Dette er under antagelsen om at rekkefølgen på ordlistene ikke spiller noen rolle (ofte vil man kanskje ha de sortert eller prioritert etter frekvens).

Å hente ut alle måter å fullføre et påbegynt ord er forventet konstant tid, ettersom vi kan returnere en liste som allerede er lagret. Dette er altså mer effektivt enn prefiks-trevarianten. Det sagt vil forslagene vises frem, og det krever lineær tid med hensyn til antall «completions».

Til slutt ender dette med å være hovedsakelig en avveining mellom tidsbruk og minnebruk. Hvis vi tar det å vise ordene med i betraktningen, så virker det som at det er lite tid å spare på det raske oppslaget i hashmapet.

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.

(a) 1 poeng. Siden vi har prioriteter fra 0 til et kjent maksimum N, passer et enkelt array av bøtter perfekt. Bøttene kan for eksempel være en enkeltlenket liste. Dette er helt tilsvarende bucket sort.

(b) Opp til 3 poeng. Legg til x i bøtta på plass p i Q.

```
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)
2  |  B ← Q[p]
3  |  add x to B
```

(c) Opp til 4 poeng. Finn det den første ikke-tomme bøtta, og fjern og returner det første i den bøtta.

```
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)
2  |  for i ← 0 to N − 1 do
3  |  |  B ← Q[i]
4  |  |  if B is not empty then
5  |  |  |  return B.pop()
```

(d) 1 poeng for hver av algoritmene. Siden N er konstant så er begge algoritmene $\mathcal{O}(1)$.

En besvarelse som tar utgangspunkt i en binær heap (eller andre mindre egnede strukturer) kan få maksimalt 4 poeng.

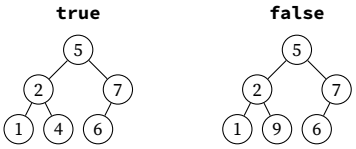
Er binærtreet et søketre? 10 poeng

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.

(a) Verdien i hver node må være *større enn alle* verdier i venstre subtre og *mindre enn alle* verdier i høyre subtre.

Det gis 2 poeng for å oppgi riktig egenskap. Spørsmålet er gitt mest for å sørge for at studenten har dette klart for seg før neste deloppgave.

(b) Her er tre «naturlige» løsninger på problemet.

En $\mathcal{O}(n^2)$ løsning som følger definisjonen av binære søketre godt kan se slik ut:

```
1 Procedure CheckBST(v)
2 |  if v = null then
3 |  |  return true
4 |  if v.element < FindMax(v.left) then
5 |  |  return false
6 |  if v.element > FindMin(v.right) then
7 |  |  return false
8 |  return CheckBST(v.left) and CheckBST(v.right)
```

Ved å regne ut minste og største noder i treet først får vi en fin $\mathcal{O}(n)$ løsning:

```
1  Procedure CheckBST(v)
2  |  return CheckBSTHelper(v, FindMin(v), FindMax(v))
3
4  Procedure CheckBSTHelper(v, low, high)
5  |  if v = null then
6  |  |  return true
7  |
8  |  if v.element < low or v.element > high then
9  |  |  return false
10 |
11 |  checkLeft ← CheckBSTHelper(v.left, low, v.element)
12 |  checkRight ← CheckBSTHelper(v.right, v.element, high)
13 |
14 |  return checkLeft and checkRight
```

En annen $\mathcal{O}(n)$ løsning er å gjøre en in-order traversering av det binære treet, og legge nodene i en liste. Dersom den resulterende listen er sortert, så er treet også et binært søketre.

```
1  Procedure CheckBST(v)
2  |  inorder_list ← InOrder(v, empty list)
3  |  for consecutive pairs (x, y) in inorder_list do
4  |  |  if x > y then
5  |  |  |  return false
6  |  return true
7
8  Procedure InOrder(v, list)
9  |  if v = null then
10 |  |  return list
11 |
12 |  InOrder(v.left, list)
13 |  append v.element to list
14 |  InOrder(v.right, list)
15 |
16 |  return list
```

Det gis opp til 4 poeng for en kvadratisk løsning, og opp til 6 poeng for en lineær løsning.

(c) Det gis 2 poeng for riktig kjøretidskompleksitet.

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).

(a) Opp til 3 poeng. Returner antall komponenter grafen består av.

(b) Opp til 3 poeng. Returner antall noder med inngrad 0.

(c) Opp til 4 poeng. Finn de sterkt sammenhengende komponentene av grafen; returner antall noder med inngrad 0 i komponentgrafen.