Om eksamensoppgavene

Emnekode og -navn: ITF20006 Algoritmer og datastrukturer

Dato og tid: 12.05.2023, varighet 4 timer
Fagansvarlig: Jan Høiberg
Tillatte hjelpemidler: Alle trykte og skrevne hjelpemidler er tillatt
Sensurfrist: 02.06.2023**,** resultater blir publisert i Studentweb.

Eksamensoppgavesettet består av:

- En innledning med informasjon om oppgavesettet (siden som du nå leser).
- Åtte eksamensoppgaver med spørsmål fra ulike deler av pensum.
- To vedlegg med Java-dokumentasjon og kode
- En avsluttende side med et kommentarfelt, der du kan legge inn evt. egne kommentarer.

De fem første eksamensoppgavene er alle flervalgsoppgaver. Hver av dem inneholder fem deloppgaver nummerert fra A til E. I hver deloppgave skal det velges *ett* riktig svar fra flere svaralternativer. Det gis *ikke* minuspoeng for feil svar i flervalgsoppgavene.

Oppgavene 6 og 7 skal besvares med tekst/tabeller. Den siste eksamensoppgaven er en programmeringsoppgave der besvarelsen skal skrives i Java. Legg vekt på å skrive enkel og ryddig kode.

Eksamensoppgavene vektes slik:

- Hver av oppgavene 1-5 teller 10%
- Oppgavene 6 og 7 teller begge 15%
- Oppgave 8 teller 20%

Alle deloppgavene innenfor en og samme eksamensoppgave vektes likt.

Merk at Java-klassen *String* for tegnstrenger brukes i flere av oppgavene. I vedlegg 1 finner du dokumentasjon av to metoder i *String*-klassen som er gitt i oppgaveteksten og/eller kan brukes i din egen kode.

1 Oppgave 1: Algoritmeanalyse

I hver av de fem deloppgavene A-E nedenfor er det beskrevet en operasjon eller beregning som skal utføres av et program. Tiden som programmet bruker på å gjøre dette vil avhenge av størrelsen på problemet, som er gitt som et positivt heltall *n*.
For hver deloppgave skal du angi den korrekte (worst-case) arbeidsmengden for programmet, angitt med *O*-notasjon, ved å velge ett av de fire alternativene som er gitt som svar.

Deloppgave A
Rekursiv beregning av *n!* (*n*-fakultet).

- $O(\log n)$
- $O(n)$
- $O(n \cdot \log n)$
- $O(n^2)$

Deloppgave B
Innsetting av en ny verdi i et AVL-tre med *n* noder.

- $O(\log n)$
- $O(\mathrm{n})$
- $O(n \cdot \log n)$
- $O(n^2)$

Deloppgave C
Innsetting av *n* verdier i en sortert lenket liste, som er tom når innsettingen starter.

- $O(\log n)$
- $O(n)$
- $O(n \cdot \log n)$
- $O(n^2)$

Deloppgave D
Innsetting av n verdier i en heap som er tom når innsettingen starter.

- $O(\log n)$
- $O(n)$
- $O(n \cdot \log n)$
- $O(n^2)$

Deloppgave E

Rehashing av alle elementene i en hashtabell som har lengde lik n.

Du kan anta følgende: Før rehashingen utføres, dobles hashlengden til $2 \cdot n$. Hashingen gjøres med åpen adressering og kvadratisk probing. Det brukes en hashfunksjon som sprer godt og gir lite kollisjoner.

- $O(n)$
- $O(n \cdot \log n)$
- $O(n^2)$
- $O(n^3)$

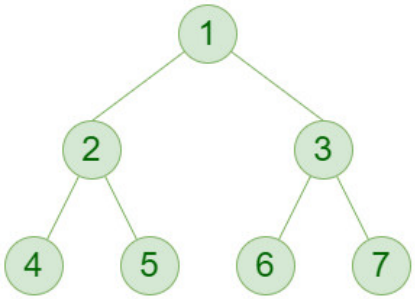
Maks poeng: 10

2 Oppgave 2: Stack og kø

I deloppgavene A-E nedenfor er det gitt kode for fem ulike metoder, som alle bruker Javas implementasjoner (fra *java.util*) av enten stack eller kø. Merk spesielt at operasjonene *enqueue* og *dequeue* på en kø gjøres med metodene *add* og *remove* i Java.
Alle de fem metodene traverserer et binært tre og skriver ut verdiene i treet. Nodene i det binære treet representeres med denne klassen:

```
class BinærNode
{
   int data;
   BinærNode venstre;
   BinærNode høyre;
}
```

Følgende binære tre er gitt:



Du skal angi rekkefølgen som verdiene i treet skrives ut i av hver av de fem metodene gitt i oppgave A-E, når de kalles med roten i treet ovenfor som parameter. Merk at noe av koden i metodene er **uthevet** for å kanskje hjelpe på forståelsen av hvordan de virker.

Deloppgave A

```
void metodeA(BinærNode rot)
{
   if (rot != null)
   {
      Queue<BinærNode> Q = new LinkedList<BinærNode>();
      Q.add(rot);
      while (!Q.isEmpty())
      {
         BinærNode node = Q.remove();
         System.out.print(node.data + " ");
         if (node.venstre != null)
            Q.add(node.venstre);
         if (node.høyre != null)
            Q.add(node.høyre);
      }
   }
}
```

I hvilken rekkefølge skrives verdiene i treet ut av *metodeA*?

- 1 2 3 4 5 6 7
- 1 2 4 5 3 6 7
- 1 3 2 7 6 5 4
- 1 3 7 6 2 5 4
- 4 2 5 1 6 3 7

Deloppgave B

```
void metodeB(BinærNode rot)
{
   if (rot != null)
   {
      Queue<BinærNode> Q = new LinkedList<BinærNode>();
      Q.add(rot);
      while (!Q.isEmpty())
      {
         BinærNode node = Q.remove();
         System.out.print(node.data + " ");
         if (node.høyre != null)
            Q.add(node.høyre);
         if (node.venstre != null)
            Q.add(node.venstre);
      }
   }
}
```

I hvilken rekkefølge skrives verdiene i treet ut av *metodeB*?

- 1 2 3 4 5 6 7
- 1 2 4 5 3 6 7
- 1 3 2 7 6 5 4
- 1 3 7 6 2 5 4
- 4 2 5 1 6 3 7

Deloppgave C

```
void metodeC(BinærNode rot)
{
   if (rot != null)
   {
      Stack<BinærNode> S = new Stack<BinærNode>();
      S.push(rot);
```

```
      while (!S.isEmpty())
      {
         BinærNode node = S.pop();
         System.out.print(node.data + " ");
         if (node.venstre != null)
            S.push(node.venstre);
         if (node.høyre != null)
            S.push(node.høyre);
      }
   }
}
```

I hvilken rekkefølge skrives verdiene i treet ut av *metodeC*?

- 1 2 3 4 5 6 7
- 1 2 4 5 3 6 7
- 1 3 2 7 6 5 4
- 1 3 7 6 2 5 4
- 4 2 5 1 6 3 7

Deloppgave D

```
void metodeD(BinærNode rot)
{
   if (rot != null)
   {
      Stack<BinærNode> S = new Stack<BinærNode>();
      S.push(rot);
      while (!S.isEmpty())
      {
         BinærNode node = S.pop();
         System.out.print(node.data + " ");
         if (node.høyre != null)
            S.push(node.høyre);
         if (node.venstre != null)
            S.push(node.venstre);
      }
   }
}
```

I hvilken rekkefølge skrives verdiene i treet ut av *metodeD*?

- 1 2 3 4 5 6 7
- 1 2 4 5 3 6 7
- 1 3 2 7 6 5 4
- 1 3 7 6 2 5 4
- 4 2 5 1 6 3 7

Deloppgave E

```
static void metodeE(BinærNode rot)
{
   if (rot != null)
   {
      Stack<BinærNode> S = new Stack<BinærNode>();
      BinærNode node = rot;
      while (node != null || !S.isEmpty())
      {
         while (node != null)
         {
            S.push(node);
            node = node.venstre;
         }
         node = S.pop();
         System.out.print(node.data + " ");
         node = node.høyre;
      }
   }
}
```

I hvilken rekkefølge skrives verdiene i treet ut av *metodeE*?

- 1 2 3 4 5 6 7
- 1 2 4 5 3 6 7
- 1 3 2 7 6 5 4
- 1 3 7 6 2 5 4
- 4 2 5 1 6 3 7

Maks poeng: 10

3 Oppgave 3: Rekursiv programmering

Deloppgave A

Nedenfor er det gitt tre metoder for å beregne summen *S(n)* av de *n* første naturlige tallene:

$$S(n) = 1 + 2 + 3 + \ldots + (n-1) + n$$

```
long metodeA1(int n)
{
   long S = 0;
   for (int i = 1; i <= n; i++)
      S += i;
   return S;
}

long metodeA2(int n)
{
   if (n == 0)
      return 0;
   return n + metodeA2(n - 1);
}

long metodeA3(int n)
{
   return (long) n * (n + 1) / 2;
}
```

Hvilken av de tre metodene er minst effektiv for store verdier av *n*?

- metodeA1
- metodeA2
- metodeA3
- Alle tre metoder er like effektive

Deloppgave B

Følgende rekursive metode er gitt (se vedlegg 1 for dokumentasjon av metoden *substring*):

```
static void metodeB(String S)
{
   int n = S.length();
   if (n > 2)
   {
      for (int i = 1; i < n - 1 ; i++)
         System.out.print(S.substring(0, 1) +
                          S.substring(i, i + 2) + " ");
      metodeB(S.substring(1, n));
   }
}
```

Hva skrives ut av metodekallet *metodeB("ABCD")* ?

- DCB DCA CBA
- ABC ACD BCD
- AAB BBC CCD
- BCD ACD ABC

Deloppgave C

Følgende to Java-metoder som kaller hverandre er gitt:

```
void metodeC1(int n)
{
   if (n == 0) return;
   System.out.print(n);
   metodeC2(n - 2);
   System.out.print(n);
}

void metodeC2(int n)
{
   if (n == 0) return;
   System.out.print(n);
   metodeC1(n + 1);
   System.out.print(n);
}
```

Hva vil skrives ut av metodekallet *metodeC1(3)* ?

- 312213
- 321123
- 302203
- 310013

Deloppgave D

Følgende Java-metode er gitt:

```
int metodeD(int m, int n)
{
   if (n == 0)
      return m;
   return (1 + metodeD(m, n - 1));
}
```

Du kan anta at parameteren *n* er et positivt heltall. Hva er arbeidsmengden til *metodeD*?

- $O(n^2)$
- $O(\log n)$
- $O(n \cdot m)$
- $O(n)$

Deloppgave E

Hva beregnes av metoden gitt i deloppgave D ovenfor, når parameteren n er et positivt heltall?

- $m \cdot n$
- m^n
- $m - n$
- $m + n$

Maks poeng: 10

4 Oppgave 4: Søking og sortering

Deloppgave A

En sortert array *A* med lengde lik 15 inneholder disse heltallene:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	4	8	10	15	18	21	24	27	29	33	34	37	39	41	43

Det skal brukes et sekvensielt søk for å søke etter en verdi *x* i arrayen. Hvor mange av tallene i *A* vil bli sammenlignet med *x* når vi søker etter *x* = 41?

- 2
- 3
- 10
- 14

Deloppgave B

Det skal brukes et binært søk for å søke etter en verdi *x* i arrayen *A* gitt ovenfor. Hvor mange av tallene i *A* vil bli sammenlignet med *x* når vi søker etter *x* = 41?

- 2
- 3
- 10
- 14

I hver av deloppgavene C,D og E nedenfor skal du velge en av fire algoritmer for å sortere en array, slik at verdiene står i stigende rekkefølge etter sorteringen. Du kan anta følgende:

- Arrayen inneholder et meget stort antall elementer, slik at sorteringen kan ta svært lang tid.
- I quicksort-algoritmen brukes alltid første element i en delarray som partisjoneringselement.

Deloppgave C

Hvilken sorteringsalgoritme er raskest når arrayen er nesten sortert, dvs. at det bare er noen få elementer som står feil plassert?

- Innstikksortering
- Utplukksortering
- Quicksort
- Heapsort

Deloppgave D

Hvilken sorteringsalgoritme er raskest når arrayen allerede er sortert, men i omvendt (avtagende) rekkefølge?

- Innstikksortering
- Utplukksortering
- Quicksort
- Heapsort

Deloppgave E

Hvilken sorteringsalgoritme er raskest når alle verdiene i arrayen er ulike og helt tilfeldig fordelt?

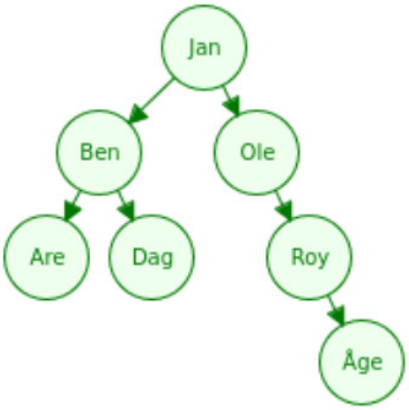
- Innstikksortering
- Utplukksortering
- Quicksort
- Heapsort

Maks poeng: 10

5 Oppgave 5: Binære søketrær

Deloppgave A

Følgende vanlige søketre (ikke AVL) med syv noder, som er alfabetisk sortert på norske bokstaver, er gitt:



Hva er bredde-først (nivå-for-nivå) rekkefølgen av verdiene i treet etter at en ny node med verdien "Kai" er satt inn?

- Jan Ben Ole Are Dag Kai Roy Åge
- Are Ben Dag Jan Kai Ole Roy Åge
- Jan Ole Ben Roy Kai Dag Are Åge
- Jan Ben Ole Are Dag Roy Åge Kai

Deloppgave B

Rotnoden med verdien "Jan" skal fjernes fra søketreet som er tegnet i Deloppgave A ovenfor. Hva er bredde-først rekkefølgen av de gjenværende seks verdiene etter at fjerningen av roten er utført?

- Dag Ben Ole Are Roy Åge
- Are Ben Dag Ole Roy Åge
- Dag Ole Ben Roy Are Åge
- Ben Are Ole Dag Roy Åge

Deloppgave C

Vi definerer høyden av et binært tre som lengden av lengste vei fra roten til en bladnode, dvs. at et tre med bare én node har høyde lik 0. Anta at følgende åtte verdier settes inn i et tomt søketre i omvendt alfabetisk rekkefølge:

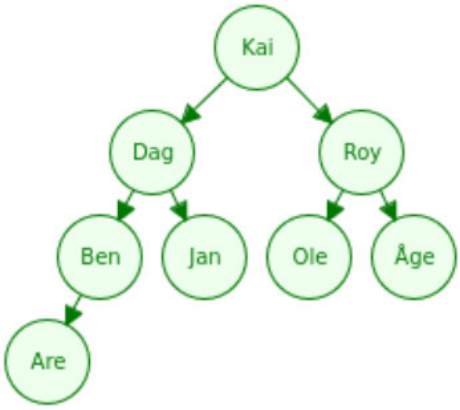
Åge Roy Ole Kai Jan Dag Ben Are

Hva er høyden av søketreet etter at alle de åtte verdiene er satt inn i denne rekkefølgen?

- 3
- 4
- 6
- 7

Deloppgave D

Følgende binære søketre er gitt:



Hva er postorder rekkefølgen av verdiene i dette treet?

- Kai Dag Roy Ben Jan Ole Åge Are
- Kai Dag Ben Are Jan Roy Ole Åge
- Are Ben Dag Jan Kai Ole Roy Åge
- Are Ben Jan Dag Ole Åge Roy Kai

Deloppgave E

Anta nå at treet som er tegnet i deloppgave D ovenfor er et AVL-tre. Hvilke to verdier ligger på det nederste nivået i dette AVL-treet etter at vi har satt inn verdien "Ali"?

- Are og Ben
- Are og Ali
- Ben og Jan
- Ali og Ben

Maks poeng: 10

6 Oppgave 6: Hashing

Hvis vi bruker hashing med åpen adressering, kan en hashtabell med lengde ti som inneholder åtte heltall fremstilles slik:

```
0: 19
1: 31
2:
3: 13
4: 63
5: 93
6:
7: 77
8:  8
9: 58
```

Hvis vi bruker kjeding (med lenkede lister) til håndtere kollisjoner, kan hashtabellen se slik ut:

```
0:
1: 31
2:
3: 93 63 13
4:
5:
6:
7: 77
8: 58 8
9: 19
```

I deloppgavene A-E nedenfor skal du vise hvordan heltallsverdier settes inn en hashtabell med bruk av ulike hashingteknikker. Hashfunksjonen som brukes er denne:

```
int hash(int verdi) { return verdi % 10; }
```

Hashindeksen som beregnes for en verdi er altså lik resten ved heltallsdivisjon med hashlengden. Følgende åtte verdier skal settes inn, i denne rekkefølgen, i en hashtabell med lengde lik ti, som til å begynne med er tom:

```
  12 34 14 25 47 38 17 27
```

Deloppgave A

Vis hvordan hashtabellen ser ut etter innsetting av de åtte verdiene når det brukes åpen addressering med vanlig lineær probing:

Deloppgave B

Vis hvordan hashtabellen ser ut etter innsetting av de åtte verdiene når det brukes åpen addressering med kvadratisk probing:

Deloppgave C

Vis hvordan hashtabellen ser ut etter innsetting av de åtte verdiene når det brukes hashing med kjeding, med lenkede lister:

Deloppgave D

Vis hvordan hashtabellen ser ut etter innsetting av de åtte verdiene når det brukes åpen adressering med "Last Come First Serve"-hashing og lineær probing:

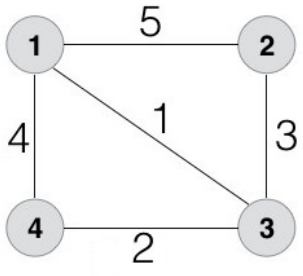
Deloppgave E

Vis hvordan hashtabellen ser ut etter innsetting av de åtte verdiene når det brukes åpen adressering med "Robin Hood"-hashing og lineær probing:

Maks poeng: 10

7 Oppgave 7: Grafer

Alle deloppgavene i oppgave 7 bruker følgende urettede og vektede graf med fire noder:



I algoritmene som brukes på denne grafen, skal alltid naboene til en node oppsøkes i stigende nummerrekkefølge. Merk at vi bruker nodeverdiene (1, 2, 3, 4) også som tabellindekser.

Deloppgave A

Hva er rekkefølgen av nodene i en dybde-først traversering med start i node 4?

Deloppgave B

Hva er rekkefølgen av nodene i en bredde-først traversering med start i node 4?

Deloppgave C

Kantlengdematrisen (der * betyr "ingen kant") for grafen som er tegnet ovenfor ser slik ut:

	1	2	3	4
1	**0**	**5**	**1**	**4**
2	**5**	**0**	**3**	*****
3	**1**	**3**	**0**	**2**
4	**4**	*****	**2**	**0**

Floyds algoritme, som kan programmeres enkelt med tre for-løkker inne i hverandre, omformer denne kantlengdematrisen til en matrise med lengden av korteste vei mellom alle par av noder i grafen. Vis hvordan matrisen gitt ovenfor ser ut etter første gjennomløp av den ytterste løkken i Floyd:

Deloppgave D

Dijkstras agoritme skal kjøres på grafen gitt ovenfor, med start i node 1.

Sett opp en oversiktlig tabell som viser hovedstegene som gjøres i algoritmen, frem til vi har funnet korteste vei til alle de andre nodene i grafen. Hver rad i tabellen skal vise tilstanden etter én iterasjon i algoritmen, og skal angi:

- Hvilke noder som vi garantert kjenner korteste vei til, og hvor lang denne veien er.
- Hvilke andre noder som er oppsøkt og som vi kjenner en mulig korteste vei til, og hvor lang denne veien er.
- Hvilke noder som ennå ikke er oppsøkt.

Skriv ditt svar her

Deloppgave E

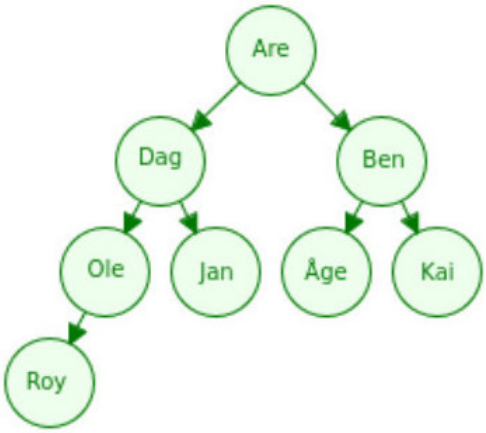
I Dijkstras algoritme kan nodene langs den korteste veien frem til en node lagres ved å bruke en array, som for hver node (untatt startnoden, der vi kan sette inn verdien -1) bare lagrer den siste forgjengeren på veien frem til noden. Hvordan vil denne forgjengerarrayen se ut for grafen ovenfor når startnoden for Dijkstra er node 1?

Skriv ditt svar her

Maks poeng: 10

8 Oppgave 8: Programmering - Heap

I denne oppgaven skal det lages Java-metoder for å håndtere en minheap der dataene er tekststrenger. Et eksempel på en slik heap, fremstilt både som et binært tre og som en array som lagrer dataene i heapen, er vist nedenfor:



Are	Dag	Ben	Ole	Jan	Åge	Kai	Roy	...
0	1	2	3	4	5	6	7	...

I vedlegg 2 er det gitt en klasse *StringHeap* som skal implementere en minheap. Det er her flere Java-metoder som ikke er laget ferdig:

```
private int venstre(int indeks)
private int høyre(int indeks)
private int forelder(int indeks)
public boolean settInn(String S)
public String fjernMinste()
```

Du skal programmere disse metodene i deloppgavene A, B og C nedenfor. I Deloppgave D skal det programmeres en metode som bruker en *StringHeap*. All Java-koden fra hver deloppgave skal skrives inn i det samme tekstfeltet som ligger nederst i oppgaven.

Deloppgave A

I en heap lagret som array, kan vi finne både de to barna til en node og nodens forelder med et enkelt indeksregnestykke. Programmer de tre metodene *venstre*, *høyre* og *forelder* i klassen *StringHeap*, som implementerer denne indeksberegningen. Merk at *venstre* og *høyre* kan returnere indekser som er utenfor heapen. Metoden *forelder* skal returnere verdien null for rotnoden.

Deloppgave B

Programmer metoden *settInn* i klassen *StringHeap*. Metoden skal sette inn en ny tekststreng i heapen. Hvis heapen er full skal metoden returnere *false* uten å gjøre noen innsetting, ellers skal den returnere *true*. Metoden *forelder* som du laget i deloppgave A skal brukes i *settInn*. Du skal også bruke metoden *compareTo* fra *String*-klassen, gitt i vedlegg 1.

Deloppgave C

Programmer metoden *fjernMinste* i klassen *StringHeap*. Metoden skal fjerne og returnere den minste verdien i heapen. Hvis heapen er tom skal metoden returnere verdien *null*. Metodene *venstre* og *høyre* som du laget i deloppgave A skal brukes i *settInn*. Du skal også bruke metoden *compareTo* fra *String*-klassen, gitt i vedlegg 1.

Deloppgave D

Programmer en metode:

```
void skrivSortert(String A[])
```

skrivSortert skal skrive ut alle strengene i arryayen A i sortert rekkefølge. Metoden skal bruke en *StringHeap* til å løse problemet. De to metodene du laget i deloppgavene B og C ovenfor skal begge kalles like mange ganger som lengden på arrayen *A*.

Merk at du kan løse denne oppgaven selv om du ikke har programmert ferdig metodene *settInn* og *fjernMinste*.

Skriv svarene på alle deloppgavene A - D her:

```
1
```

Maks poeng: 15

Vedlegg 1: To metoder for tekststrenger i Java

Class String Method Summary

`public int compareTo(String anotherString)`

Compares two strings lexicographically. Returns the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.

`public String substring(int beginIndex, int endIndex)`

Returns a string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex-1`. Thus the length of the substring is `endIndex-beginIndex`.

Examples: `"hamburger".substring(4, 8)` returns `"urge"`
`"smiles".substring(1, 5)` returns `"mile"`

Vedlegg 2: Klassen StringHeap

```
public class StringHeap
{
   private String heap[]; // Array som lagrer heapen
   private int max;       // Lengden av heaparrayen
   private int min = 100; // Minste tillatte arraylengde
   private int n;         // Antall strenger lagret i heapen

   // Konstruktør, oppretter tom heap med gitt lengde
   public StringHeap(int lengde)
   {
      max = lengde < min ? min : lengde;
      heap = new String[max];
      n = 0;
   }

   // Returnerer indeks til venstre barn for en node
   private int venstre(int indeks)
   {
      <Skal programmeres i oppgave 8 A>
   }

   // Returnerer indeks til høyre barn for en node
   private int høyre(int indeks)
   {
      <Skal programmeres i oppgave 8 A>
   }

   // Returnerer indeks til forelder til en node
   private int forelder(int indeks)
   {
      <Skal programmeres i oppgave 8 A>
   }

   // Setter inn ny verdi i heapen
   public boolean settInn(String S)
   {
      <Skal programmeres i oppgave 8 B>
   }

   public String fjernMinste()
   {
      <Skal programmeres i oppgave 8 C>
   }
}
```

Egne kommentarer

Her kan du legge inn evt. kommentarer til oppgavene og til din egen besvarelse.