Høgskolen i Østfold

EKSAMEN

Emnekode:	Emnenavn:
ITF20006	Algoritmer og datastrukturer
Dato:	**Eksamenstid:**
18. mai 2017	4 timer
Hjelpemidler:	**Faglærere:**
Alle trykte og skrevne Kalkulator	Jan Høiberg

Om eksamensoppgaven og poengberegning:

Oppgavesettet består av 10 sider inkludert denne forsiden og de 4 vedleggene.

Kontroller at oppgaven er komplett før du begynner å besvare spørsmålene.

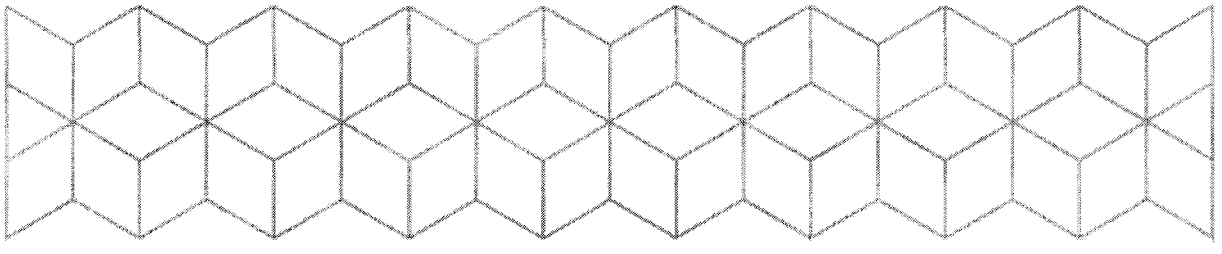
Oppgavesettet består av 4 oppgaver med i alt 20 deloppgaver. Innen hver oppgave vektes alle deloppgaver likt.

Les oppgavetekstene nøye før du begynner på besvarelsen.

Legg vekt på å skrive en ryddig og lett forståelig besvarelse.

Sensurfrist: 9. juni 2017

Karakterene er tilgjengelige for studenter på Studentweb senest 2 virkedager etter oppgitt sensurfrist. www.hiof.no/studentweb



Oppgave 1: Algoritmeanalyse, rekursjon, stack og kø (25%)

I deloppgavene a) – e) nedenfor skal du lage *fem* ulike versjoner av denne Java-metoden:

```
void snu(int A[])
```

Metoden `snu` skal snu om på (reversere) rekkefølgen av innholdet i arrayen (tabellen) `A`. F.eks. skal kjøring av denne Java-koden:

```
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
snu(A);
for (int i = 0; i < A.length; i++)
     System.out.print(A[i] + " ");
System.out.println();
```

resultere i denne utskriften:

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

a) Lag en *iterativ* versjon av metoden `snu` som bruker en *ekstra array* `B`. Metoden skal først kopiere innholdet i `A` over i `B`, og deretter legge innholdet tilbake i `A` i reversert rekkefølge.

b) Lag en iterativ versjon av metoden `snu` som *ikke* bruker noen ekstra array til å utføre reverseringen.

c) Lag en versjon av metoden `snu` som bruker en *rekursiv* metode `snuRek` for å utføre reverseringen av innholdet i `A`. Du skal programmere begge metodene, og må selv bestemme hvilke parametre metoden `snuRek` skal ha.

d) Lag en versjon av metoden `snu` som bruker en *stack* til å utføre reverseringen. Innholdet i `A` skal først kopieres over på stacken og deretter legges tilbake i reversert rekkefølge. Bruk metoder fra klassen `intStack` som er gitt i vedlegg 1 til å håndtere stacken.

e) Lag en versjon av metoden `snu` som bruker en *vanlig kø* til å utføre reverseringen. Innholdet i `A` skal først legges inn i køen og deretter legges tilbake i reversert rekkefølge. Bruk metoder fra klassen `intQueue` som er gitt i vedlegg 2 til å håndtere køen.

I oppgavene f) og g) nedenfor kan du la verdien n betegne antall elementer i arrayen `A`.

f) Alle metodene i deloppgavene a) – e) har arbeidsmengde av samme størrelsesorden. Hva er arbeidsmengden, uttrykt med n og O-notasjon? Gi en kort begrunnelse for svaret.

g) Hvor mye *ekstra primærminne* (RAM), i *tillegg* til det som trengs for å lagre arrayen `A`, krever *hver* av metodene du har laget i deloppgavene a) – e), uttrykt med n og O-notasjon? Gi en kort begrunnelse for svarene.

(Slutt på oppgave 1)

Oppgave 2: Søking og sortering (15%)

a) I denne oppgaven skal du lage følgende Java-metode:

```
int finn(int A[], int verdi)
```

Parameteren `A` er en array med heltall som er *sortert* i stigende rekkefølge. Metoden skal søke i `A` etter verdien angitt i parameteren `verdi`. Merk at det kan være flere like verdier i `A`.

Metoden `finn` skal virke på denne måten:

- Hvis `verdi` finnes i arrayen `A`, skal metoden returnere *indeksen* i `A` der denne verdien ligger lagret. Indeksen er heltall større eller lik null og mindre enn antall elementer i arrayen.
- Hvis `A` inneholder *flere* forekomster av `verdi`, skal metoden returnere indeksen til *første* forekomst.
- Hvis `verdi` *ikke* finnes i `A`, skal metoden returnere et *negativt* tall som er lik:

  ```
  - (index + 1)
  ```

  der `index` er indeksen i `A` der verdien *ville* ha ligget dersom den hadde vært lagret i array.

Her er et Java-eksempel som viser hvorledes metoden `finn` skal fungere:

```
int A[]  = {2, 3, 5, 7, 10, 11, 12, 12, 18, 20};
System.out.println( finn(A,  1) ); // Utskrift: -1
System.out.println( finn(A,  3) ); // Utskrift: 1
System.out.println( finn(A, 12) ); // Utskrift: 6
System.out.println( finn(A, 16) ); // Utskrift: -9
System.out.println( finn(A, 21) ); // Utskrift: -11
```

Den beste løsningen her er å lage metoden `finn` med en algoritme som ligner på binærsøk, men det kan også brukes andre og enklere teknikker.

b) I vedlegg 3 er det gitt en Java-metode:

```
int metode_2b(int A[])
```

1. Hva utføres av denne metoden?
2. La *n* betegne antall elementer i arrayen `A`. Hva er arbeidsmengden til `metode_2b`, angitt med *n* og *O*-notasjon?

(Slutt på oppgave 2)

Oppgave 3: Binære søketrær (35%)

Følgende 14 heltall er gitt i denne rekkefølgen:

```
10  5  9  1  15  13  4  6  8  2  14  11  12  3
```

a) Tallene skal settes inn i et vanlig *binært søketre* som i utgangspunktet er tomt. Tegn en figur som viser hvorledes søketreet ser ut etter at alle de 14 verdiene er satt inn i treet.

b) Den vanlige algoritmen for fjerning av en node med en gitt verdi i et søketre, skal brukes på treet du tegnet i deloppgave a). Verdien 1 fjernes først fra dette treet, deretter fjernes verdien 10. Tegn en figur som viser hvorledes treet ser ut etter at begge disse to verdiene er fjernet.

I vedlegg 4 er det gitt en Java-klasse `intSearchTree` for binære søketrær, der dataene i hver node er et enkelt, positivt heltall. I deloppgavene nedenfor skal du bl.a. programmere noen metoder for denne klassen.

c) Forklar kort hvor du *alltid* finner den minste verdien i et binært søketre.

d) Lag en Java-metode:

```
public int min()
```

i klassen `intSearchTree` , som returnerer den minste verdien i treet. Hvis treet er tomt, returneres verdien 0 (null).

Vi definerer *høyden* av et binært tre som lengden av den *lengste* veien i treet, fra roten og ut til en bladnode. Dette er det samme som antall nivåer i treet, minus én. Det betyr f.eks. at et binært tre med bare én (rot)node har høyde lik 0 (null) og et tre med to noder har høyde lik 1.

e) Hva er høyden av søketreet som du tegnet i deloppgave a)?

f) Metoden `insert` i klassen `intSearchTree` setter inn en ny verdi i søketreet på vanlig måte. Klassen inneholder en variabel `height` som lagrer høyden av søketreet. Skriv om metoden `insert` slik at verdien til `height` settes riktig ved innsetting av nye verdier.

I de to siste deloppgavene i oppgave 3, skal vi se på nodene på det *nederste* nivået i søketreet.

g) Hvilke verdier ligger på nederste nivå i de to søketrærne du tegnet i hhv. deloppgave a) og deloppgave b)?

h) Lag en Java-metode:

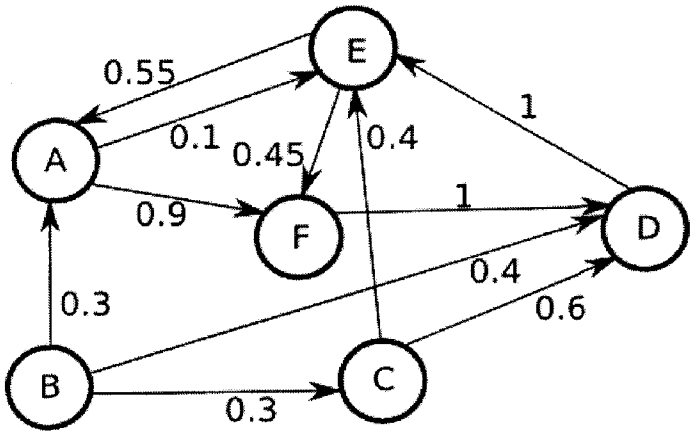
```
public void lowestLevel()
```

i klassen `intSearchTree` , som skriver ut *alle* verdiene på nederste nivå i søketreet, i sortert rekkefølge. Ingen andre verdier i treet skal skrives ut. Løsningen din *skal* bruke rekursjon og skal *ikke* bruke noen ekstra array eller kø.

(Slutt på oppgave 3)

Oppgave 4: Grafer (25%)

I denne oppgaven brukes følgende rettede, vektede graf med 6 noder:



Nodene nummereres fra 0 til 5 på denne måten: A → 0 B → 1 C → 2 D → 3 E → 4 F → 5

a) Tegn *nabomatrisen* med kantlengder for denne grafen.

Ikke-eksisterende kanter mellom to noder kan markeres i matrisen enten som "blanke" (for å spare skriving) eller ved å bruke f.eks. symbolene * ("stjerne") eller ∞ ("uendelig").

b) Floyds algoritme løser problemet med å finne korteste vei mellom alle par av noder i en graf, ved å gjøre om nabomatrisen til en matrise med veilengder. Tegn løsningsmatrisen for Floyds algoritme for grafen gitt ovenfor.

Merk at du *ikke* skal lage tegninger som viser de enkelte stegene i Floyds algoritme, men *bare* løsningen som inneholder lengden på de korteste veiene.

c) Sett opp en oversiktlig tabell som viser de 6 hovedstegene som gjøres i Dijkstra's algoritme for grafen ovenfor, med start i node B, helt 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.

(Slutt på oppgave 4)

Vedlegg 1

Enkel implementasjon av stack med heltallsarray. Det gjøres ikke noen feilsjekking ved innsetting og fjerning av data.

```
public class intStack
{
    private int top, max;
    private int stack[];

    public intStack(int length)
    {
        top = 0;
        max = length;
        stack = new int[max];
    }

    public void push(int value)
    {
        stack[top] = value;
        top++;
    }

    public int pop()
    {
        top--;
        return(stack[top]);
    }

    public int peek() { return(stack[top - 1]); }

    public boolean isEmpty() { return (top == 0); }

    public int size() { return top; }
}
```

Vedlegg 2

Enkel liste-implementasjon av kø med heltall. Det gjøres ikke noen feilsjekking ved innsetting og fjerning av data.

```
public class intQueue
{
    private class node
    {
        private int value;
        private node next;

        private node(int v)
        {
           value = v;
           next = null;
        }
    }

    private node head, tail;
    private int length;

    public intQueue()
    {
        head = tail = null;
        length = 0;
    }

    public void enqueue(int v)
    {
        if (head == null)
            head = tail = new node(v);
        else
        {
            tail.next = new node(v);
            tail = tail.next;
        }
        length++;
    }

    public int dequeue()
    {
        int v = head.value;
        head = head.next;
        length--;
        return v;
    }

    public boolean isEmpty() { return (head == null); }

    public int size() { return length; }

}
```

Vedlegg 3

```
public static void metode_2b(int A[])
{
    int n = A.length;
    int step = n/2;

    while (step > 0)
    {
        for (int i = step; i < n; i++)
        {
            int j = i;
            int temp = A[i];

            while (j >= step && A[j - step] > temp)
            {
                A[j] = A[j - step];
                j = j - step;
            }
            A[j] = temp;
        }

        if (step == 2)
            step = 1;
        else
            step /= 2;
    }
}
```

Vedlegg 4

Enkel implementasjon av binært søketre med heltall

```
public class intSearchTree
{

    // Indre klasse for nodene i treet

    private class node
    {
        private int value;
        private node left, right;

        private node(int v)
        {
            value = v;
            left = right = null;
        }
    }


    private node root;      // Roten i søketrett
    private int numNodes;   // Antall noder lagret i hele treet
    private int height;     // Høyden på hele søketreet


    // Konstruktør for å opprette et tomt søketre

    public intSearchTree()
    {
        root = null;    // Roten er null i et tomt tre
        numNodes = 0;   // Et tomt tre har ingen noder
        height = -1;    // Vi regner et tomt tre for å ha høyde lik -1
    }


   // Funksjoner for å hente ut data om søketreet

    public int numNodes() { return numNodes; }
    public int height() { return height; }
    public boolean isEmpty() { return numNodes == 0; }


    // Minste verdi i treet

    public int min()
    {
        // Skal lages i oppgave 3 d)
    }
```

(Vedlegg 4 fortsetter på neste side)

```
    // Innsetting av ny verdi i søketre, skal brukes i oppgave 3 f)

    public void insert(int value)
    {
        numNodes++;

        if (root == null)
        {
            root = new node(value);
            return;
        }

        node current = root, parent = null;

        while (current != null)
        {
            parent = current;
            current = value < parent.value ? parent.left : parent.right;
        }

        current = new node(value);
        if (value < parent.value)
            parent.left = current;
        else
            parent.right = current;
    }


    // Utskrift av nederste nivå i treet

    public void lowestLevel()
    {
        // Skal lages i oppgave 3 h)
    }

}
```