UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i	INF1020 — Algoritmer og datastrukturer
Eksamensdag:	15. desember 2004
Tid for eksamen:	14.30 – 17.30
Oppgavesettet er på 6 sider.	
Vedlegg:	Ingen
Tillatte hjelpemidler:	Alle trykte og skrevne

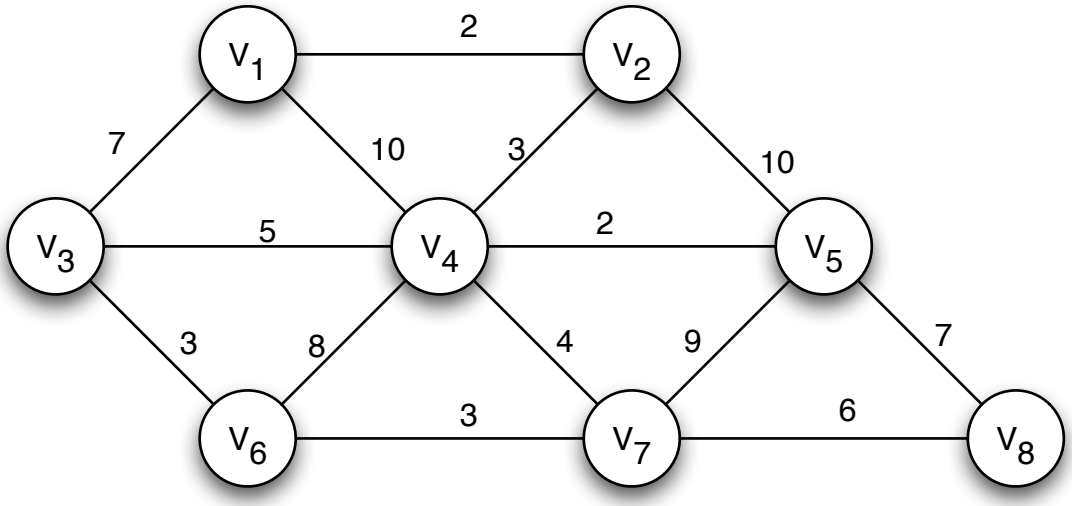
Kontroller at oppgavesettet er komplett før
du begynner å besvare spørsmålene.

Oppgave 1 Minste spenntre (15%)

Prims algoritme eksekverer ut fra en gitt startnode og velger suksessivt en ny node inntil alle noder er behandlet. Se på grafen under og bruk node v_4 som startnode. Lag så

a) en tegning av minste spenntre for grafen

b) en liste av nodene i grafen, listet opp *i den rekkefølge de har blitt valgt av Prims algoritme.*



(Fortsettes på side 2.)

Oppgave 2 Heap (85%)

I denne oppgaven vil det refereres til `class BinTree` og til skjelettet til `class DynamicHeap`. Begge disse finner du bakerst i oppgaveteksten. Legg merke til at variable for enkelhets skyld er deklarert slik at vi har tilgang til dem fra andre klasser (siden de ikke er deklarert `private`). Der det er hensiktsmessig, bør du bruke metodene i klassene, men du står selvsagt fritt til å legge til nye metoder dersom du ønsker det. Du kan også overalt og uten nærmere forklaring bruke norske metodenavn og variabelnavn istedenfor de engelske som står i oppgaveteksten.

Oppgave 2–a (5%)

Dette spørsmålet skal besvares kun med figurer. Anta at du skal bygge opp en heap av heltall. Som i pensumboken skal det være det *minste* elementet i heapen som ligger i rotnoden til heapen.

a) Tegn en figur som viser hvordan heapen ser ut etter at du har satt inn følgende tall i angitt rekkefølge i en i utgangspunktet tom heap: 20, 8, 7, 5, 10, 2.

b) Skisser hvordan heapen ser ut etter at du så har brukt `deleteMin`-operasjonen to ganger.

Oppgave 2–b (5%)

Vi vet fra pensum at en binær heap er definert som et binærtre som oppfyller et ordningskrav og et strukturkrav. Ordningskravet er knyttet til dataverdiene i nodene, mens strukturkravet uttrykker krav til treet.

a) Skriv ned ordningskravet og strukturkravet.

b) Skriv ned en formel som angir høyden til en heap som funksjon av antall elementer i heapen.

c) Vi ønsker nå å estimere hvor mange elementer det er i en heap med høyde h. Skriv ned en formel som angir et minimum antall noder i en heap med høyde h og en formel for maksimalt antall noder i en heap med høyde h.

Oppgave 2–c (10%)

I denne oppgaven skal du arbeide med binærtrær representert ved klassen `BinTree` bak i oppgaveteksten. Merk at en node i `BinTree` har en referanse til foreldrenoden i tillegg til venstre og høyre barn. Oppgaven din nå er å skrive kode for en metode

(Fortsettes på side 3.)

```
int countLessThan( BinTree t, Comparable c )
```

Vi krever at `t` skal være slik at et kall til `isHeapOrdered( t )` returnerer `true` (jvf. oppgave 2–d). Metoden skal returnere *antall* noder i `t` som har verdi mindre enn `c`. Legg vekt på å besøke så få noder som mulig!

Oppgave 2–d (15%)

Lag en metode med signatur

```
boolean isHeapOrdered( BinTree t )
```

som skal ta inn som parameter et `BinTree`-objekt `t`. Vi krever at `t` er et binærtre, dvs. at det er sykelfritt og har en unik rot, men vi krever ikke at det er en heap. Metoden skal sjekke om ordningskravet til en heap er oppfylt for `t`.

Oppgave 2–e (20%)

Du skal i denne oppgaven arbeide med en implementasjon av en heap basert på klassen `BinTree` (istedenfor den mer vanlige array-implementasjonen vi kjenner fra pensum). Skissen til heap-klassen er gitt bakerst i oppgaveteksten i skjelettet til `class DynamicHeap`. Datastrukturen til `DynamicHeap` består av et `BinTree`-objekt `root`, som representerer selve heapen. Videre skal variabelen `current` referere til den noden i heapen som ligger lengst til høyre i nederste nivå. Videre har vi en variabel som lagrer størrelsen til heapen. For å få full score på resten av oppgaven kan du ikke innføre noen ytterligere global datastruktur i `class DynamicHeap`.

I implementasjonen av `insert` og `deleteMin` skiller vi ut to hjelpemetoder som kun har til oppgave å sette inn en ny node/ta ut en node slik at strukturkravet bevares. Metoden

```
private BinTree insertNewNode( Comparable c ){ ... }
```

skal sørge for å sette inn en ny node i heapen slik at strukturkravet tilfredsstilles. Den må da settes inn til høyre i nederste nivå, evt. lengst til venstre i et nytt nivå hvis nederste nivå er fullt før den nye noden settes inn. Metoden returnerer en referanse til den nye noden i treet som har fått dataverdi `c`. Merk at etter et kall på `insertNewNode` vil det være en node mer i treet, men siden vi ikke har sjekket dataverdien til den nye noden mot noen andre verdier i treet, er treet nå generelt ikke lenger en heap.

Metoden

```
private BinTree removeLastNode(){ ... }
```

gjør det motsatte: den returnerer noden som `current` refererer til, fjerner den fra treet, og oppdaterer `current`. Oppgaven din er å kode en av disse to metodene og beskrive eller grovskissere hvordan du ville kodet den andre.

(Fortsettes på side 4.)

Oppgave 2–f (15%)

Du skal nå kode en av de to gjenstående metodene, dvs. `insert` eller `deleteMin`, og beskrive/grovskissere hvordan du ville kodet den andre. Metoden din skal kalle en av metodene fra oppgave 2–e.

Oppgave 2–g (15%)

Vi ønsker nå å legge til følgende metode til `class DynamicHeap`:

```
static DynamicHeap merge( DynamicHeap heap1, DynamicHeap heap2 )
```

Metoden skal lage en ny heap ved å slå sammen de to heapene den får inn som parameter. En fordel med å bruke en dynamisk datastruktur for heap er at denne representasjonen tillater en mer effektiv koding av merge-operasjonen enn den mer vanlige array-representasjonen av heap.

Skriv *pseudokode* for `merge`-metoden og angi dens kompleksitet. (Hint: Det lønner seg å først finne en smart løsning for tilfellene (i) begge heaper har samme høyde og den ene har nederste nivå helt fullt og (ii) den ene har nederste nivå helt fullt og den andre har ett nivå mer.) Har du dårlig tid, kan du få godt betalt for bare å angi ideen i et par linjer tekst.

(Fortsettes på side 5.)

Klassen BinTree

```
public class BinTree implements Comparable{

    Comparable data;
    BinTree left, right, parent;

    BinTree( Comparable d ){
      data=d;
      left = right = parent = null;
    }

    boolean isRoot() {
      return( parent == null );
    }

    boolean isLeftChild() {
      if( isRoot() ) return false;
      return( this == parent.left );
    }
    boolean isRightChild() {
      if( isRoot() ) return false;
      return( this == parent.right );
    }

    boolean hasLeftChild(){
      return( left != null );
    }
    boolean hasRightChild(){
      return( right != null );
    }

    BinTree getLeftMost(){
      if( hasLeftChild() ) return left.getLeftMost();
      return this;
    }
    BinTree getRightMost(){
      if( hasRightChild() ) return right.getRightMost();
      return this;
    }

    public int compareTo( Object o){
      BinTree t = (BinTree) o;
      return data.compareTo( t.data );
    }
}
```

(Fortsettes på side 6.)

Skjelett til klassen DynamicHeap

```
class DynamicHeap {

    BinTree root = null;
    BinTree current = null;
    int size = 0;

    int size(){ return size; }

    boolean isEmpty(){ return( root == null ); }

    private BinTree insertNewNode( Comparable c ){ ... }

    private BinTree removeLastNode(){ ... }

    private void swapData( BinTree t1, BinTree t2 ){
        Comparable temp = t1.data;
        t1.data = t2.data;
        t2.data = temp;
    }

    void insert( Comparable c ){ ... }

    Comparable deleteMin(){ ... }
}
```