Institutionen för
Datavetenskap
Chalmers, Göteborgs universitet

VT08
TDA416 (TDA415), DIT721 (INL080)
12/3-08

T E N T A M E N för

Datastrukturer och algoritmer IT, 7.5p, TDA416
Datastrukturer och algoritmer, 7.5 p, DIT721.
(Datastrukturer IT, 6 p, TDA415)
(Datastrukturer för datalingvister, 7.5 p, INL080)

DAG : 15 mars 2008 Tid : 8.30-13.30 SAL : M

Ansvarig	:	Bror Bjerner, tel 772 10 29, 55 54 40
Resultat	:	Senast 31/3 -08
Hjälpmedel	:	Häftet med Fakta-rutor, papper och penna.
Betygsgränser CTH	:	3 = 28(20) p, 4 = 39(35) p, 5 = 50 p
Betygsgränser GU	:	Godkänt = 28 p, Väl godkänt = 48 p

OBS ! För att få betyg 3/godkänt eller bättre krävs minst 10 poäng på både A- och B-delen
(gäller ej för TDA415)
(Om du endast tentar en del måste du lämna in senast **11.30**)

OBSERVERA NEDANSTÅENDE PUNKTER

- Börja varje ny uppgift på nytt blad.
- Skriv personnummer på varje blad.
- Använd bara ena sidan på varje blad.
- Klumpiga, komplicerade och/eller oläsliga delar kan ge poängavdrag.
- **L y c k a t i l l ! ! !**

Del A

Datastrukturer på abstrakt nivå.

Uppg 1: **a)** Stoppa in 'värdena' 1, 5, 4, 3 och 2 i ett binärt sökträd. Du skall stoppa in dem i **exakt** denna ordning (dvs 1 först, sedan 5, sedan 4 osv) och utför en sökning med splay-balansering efter 2. Visa de balanseringar som utförs vid sökningen genom att visa trädet du har före, med noderna som ingår i balanseringen markerade, och efter varje rotation.

Obs! inga balanseringar vid insättningen, bara vid sökning.

(3 p)

b) Stoppa in samma noder i samma ordning i ett AVL-träd och redovisa när obalanserna uppstår och hur balanseringarna sker.

(3 p)

c) Stoppa in samma noder i samma ordning i ett rödsvart träd, där insättning sker enligt 'top-down med flip'. Markera svarta noder med en fyrkant och röda noder med en cirkel. Markera alla 'flip' och rita om trädet efter varje rotation.

(3 p)

Uppg 2: Vilken eller vilka av följande påstående är sann(a) och vilken eller vilka är falsk(a). Rätt svar ger 1 poäng. Felaktigt svar ger -0.6 poäng. (Totala poängsumman blir dock minst 0 poäng).

a) Urvalssorteringens (selection sort) komplexitet beror på hur elementen är ordnade i fältet.

b) Det räcker med en enkellänkad lista för att implementera en kö effektivt.

c) Insättningssortering (insertion sort) och snabbsortering (quick sort) har samma **värstafalls**-komplexitet

d) Insättningssortering (insertion sort) och snabbsortering (quick sort) har samma **bästafalls**-komplexitet

e) Alla träd kan implementeras som ett binärt träd.

f) I en sammanhängande oriktad graf kan en kortaste väg mellan två noder alltid hittas med hjälp av Dijkstra's algoritm.

g) I en dubbellänkad lista tar det konstant tid att ta bort sista elementet.

h) I en hashtabell tar det alltid konstant tid att slå upp värdet till en nyckel.

. **(8 p)**

Uppg 3:

80	120	110	10	30	60	40	50	100	90	70	20

Visa hur sammansmältningssortering (merge sort) fungerar genom att sortera fältet ovan. Du skall göra sammansmältningen ända ner i botten och inte byta till någon annan sorteringsmetod vid någon viss storlek. Du får välja antingen rekursivt eller iterativt tankesätt, bara du är konsekvent. Givetvis får du utföra de olika sammansmältningarna parallellt i varje steg.

(5 p)

Uppg 4: **a)** Givet en riktad viktad graf, en startnod och en maximal totalvikt. Ge en algoritm, (ej kod, det skall du göra i sista uppgiften i del B), som ger alla noder som kan nås från startnoden utan att totalvikten av vägen överskrider den givna totala maximala totalvikten.

(7 p)

Vad är värstafallskomplexiteten för din algoritm ?

(1 p)

Del B

Datastrukturer på implementeringsnivå.

Uppg 5: **Notera följande:** För att göra det enkelt, så definieras metoderna och din privata klass för nedanstående uppgifter direkt i trädklassen (se nästa sida i häftet), som då ger dig tillgång till typen `TreeNode`. Vidare får du givetvis använda dig av stackar och/eller köer.

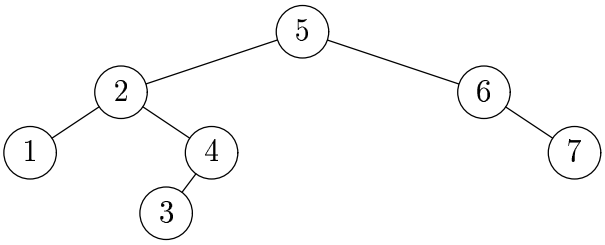
a) Givet ett binärt träd skall du göra en metod som som avgör hur många *unära interna noder* trädet innehåller. En nod är unärt intern om och endast om den har **exakt** ett delträd som inte är tomt. För full poäng krävs att komplexiteten är så låg som möjligt.

Tips: Använd rekursion !

(3 p)

b) Vad är komplexiteten för din(a) metod(er), motivera. **(1 p)**

c) Givet ett binärt träd skall du definiera en intern iterator i Java som ger noderna enligt **bredden först**. Till exempel, för trädet:



skall iteratorn ge elementen i följande ordning: 5, 2, 6, 1, 4, 7, 3

Obs! du behöver **inte** implementera metoden `remove`, bara `hasNext` och `next`. Däremot måste den givetvis definieras.

För att få full poäng skall konstruktorn, `hasNext` och `next` alla vara av $\mathbf{0}(1)$.

(10 p)

```
public class BinTree<E>
       implements Iterable<E> {

    protected TreeNode<E> root;

    public static class TreeNode<E> {

       public E           element;
       public TreeNode<E> left,
                          right;

       public TreeNode(E element) {
          this( null, element, null );
       } // constructor TreeNode

       public TreeNode( TreeNode<E> left,
                        E           element,
                      TreeNode<E>   right   ) {
            this.left    = left;
            this.element = element;
            this.right   = right;
       } // constructor TreeNode

   }  // class TreeNode

   private class BinTreeIterator
           implements Iterator<E> {

     //  din implementering  av 5.c

   } // BinTreeIterator

   // Konstruerare och metoder som redan finns
    ...

   public int noOfUnaryNodes() {
    //   din metod och ev hjälpmetod(er)
    //   till  uppgift 5.a
} // class BinTree
```

Uppg 6: Givet grafimplementeringen i lab 2 (se nedan) skall vi definiera en metod som traverserar den aktuella grafen enligt *djupet först*. Då det gäller att välja mellan grannarna för nästa nod att besöka finns inga restriktioner, utan du kan ta vilken som helst av dem. Se bara till att algoritmen terminerar och att varje nod bara besöks en gång.

Skriv en metod i Java, som utgående från en startnod ger en lista med nodnummer i den ordning som traverseringen sker. Noder som ej kan nås från den givna startnoden skall ej ingå i traverseringen. Metodhuvudet blir:

```
public List<Integer> traversDF( int startNode )
```

(6 p)

```
public abstract class Edge {

  public final int from,
                   to;

  public Edge( int from, int to ) {
    this.from = from;
    this.to  = to;
  }

  public abstract double  weight();

} // Edge
```

```
public class DirectedGraph<E extends Edge> {

  private List<E>[] neighbours;

  public DirectedGraph(int noOfNodes) {
    neighbours = (List<E>[]) new List[noOfNodes];
    for ( int i = 0; i < noOfNodes ; i++ )
      neighbours[i] = new LinkedList<E>();
  } // DirectedGraph

  public void addEdge(E e) {
    try { neighbours[e.from].add(e); }
    catch (IndexOutOfBoundsException iobe)
      { throw new IndexOutOfBoundsException(
          "Wrong Node Number in Graph: "
          + e.from); }
  } // addEdge

  public List<Integer> traversDF(int startnode){

   ***   Här skriver du din kod ***

  } // traversBF

} // class DirectedGraph
```

Uppg 7: I samma grafimplementeringen som i förra uppgiften (6), skall du också implementera algoritmen i uppgift 4.

Metodhuvudet skall definieras:

```
public Iterator<Integer>
       reachable(int startnode, double totalWeight)
```

(10 p)