Institutionen för
Datavetenskap
Göteborgs universitet, Chalmers

VT11
DIT960 (TDA416)
23/5-11

T E N T A M E N för

Datastrukturer DV, 7.5p, DIT960 (TDA416)

DAG : 30 maj 2011 Tid : 8.30-13.30 SAL : V

Ansvarig	:	Bror Bjerner, tel 772 10 29, 55 54 40
Resultat	:	senast den 7/6 -11 fås via mail med info om granskning. Lösningar på kurshemsidan efter tentan
Hjälpmedel	:	Häftet med Fakta-rutor, papper och penna.
Betygsgränser GU	:	Godkänt = 28 p, Väl godkänt = 48 p

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

OBSERVERA NEDANSTÅENDE PUNKTER

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

Del A

Datastrukturer på abstrakt nivå.

Uppg 1: **a)** Stoppa in 'värdena' 6, 1, 2, 5, 3 och 4 i ett binärt sökträd. Du skall stoppa in dem i **exakt** denna ordning (dvs först 6, sedan 1, sedan 2, sedan 5 osv) och utför en sökning med splay-balansering efter 4. 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.Visa varje 'flip' och rita om trädet efter varje rotation.

(4 p)

Uppg 2: Vilken eller vilka av följande påstående är sann(a) och vilken eller vilka är falsk(a). Svara bara med Sant eller Falskt. För varje delfråga ger rätt svar 1 p och fel svar ger -0.6 p. Poängen för hela uppgiften kan däremot inte bli negativ.

a) Värstafallskomplexiteten för sökning i ett Splay-träd är $\mathbf{O}(\,^2log\, n\,)$

b) Maximala djupet i en heap är $\lceil\,^2log\, n\,\rceil$, där n är antalet element i heapen.

c) Binär sökning är en effektiv metod för att söka i ett fält.

d) I en sammanhängande oriktad graf kan ett 'minimum spanning tree' alltid hittas med hjälp av Prim's metod.

e) För att implementera en stack med en enkellänkad struktur behövs en referens både till första och sista element i den länkade strukturen.

f) Urvalssorterings (selection sort) komplexitet beror på hur elementen är ordnade i fältet.

(6 p)

Uppg 3:

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

Visa hur 'quick sort' fungerar genom att sortera fältet ovan med den förenklade metoden, som gavs på en föreläsning. Pivotelementet skall vara **det högraste elementet** i delsegmentet som skall sorteras. Markera med bågar vilka element som byter plats och med en liten pil pivotelementen. Rita ett nytt fält efter varje pivottering. Det är givetvis tillåtet (och lämpligt) att utföra pivotteringarna av delfälten 'samtidigt' på varje nivå. Notera att du inte skall byta sorteringsmetod utan göra 'quick sort' ända ner i botten. Vidare skall sorteringen göras i fältet.

(8 p)

Uppg 4: Antag att du vill lagra heltal så att du kan söka efter dem på ett effektivt sätt. För detta ändamål kan du använda dig av någon form hashtabell.

För att illustrera hur denna datastruktur fungerar får du nu i uppgift att lagra följande tal i angiven ordning:

12, 44, 13, 88, 23, 94, 11, 39, 20, 16 och 5

a) Antag först att du lagrar talen i en hashtabell med storleken 11 och använder hashning med *hinkar* (även kallat chaining eller buckets). Din hashfunktion är

$$h(i) \quad = (2 * i \; + \; 5) \, mod \; 11$$

(*mod* är % i Java).

Rita den hashtabell du får genom att sätta in talen i angiven ordning.

(3 p)

b) I den andra varianten ska du i stället använda öppen adressering när kollision uppstår. Använd samma hashfunktion som ovan och använd den enklaste varianten där man väljer nästa lediga cell ($mod\, 11$). Rita upp tabellen, och motivera varför elementet 11 hamnar på sin plats.

(3 p)

Del B

Datastrukturer på implementeringsnivå.

Uppg 5: **a)** Du skall implementera en metod, som från en given nod ger en lista av alla (nåbara) noder enligt **bredden** först oberoende av vikten för bågarna. Ordningen mellan jämbördiga noder, dvs **obesökta grannar till en och samma nod**, ges av nodnumret, dvs lägre nodnummer kommer före högre nodnummer.

Grafen skall vara implementerad via en granntabell enligt koden nedan.

(10 p)

b) Vad blir värstafallskomplexiteten för din algoritm ?

(2 p)

```
public abstract class Edge {
  public final int from, to;
  public abstract double  weight();
} // Edge

public class DirectedGraph<E extends Edge> {

 private  List<E>[] neighbours;

 public DirectedGraph( int noOfNodes ) { .. }
 } // constuctor DirectedGraph

 public void addEdge( E e ) { .. }

 ...

 public List<Integer> breadthFirst( int startNode ) {
    /** HÄR SKRIVER DU DIN KOD  **/
 } // breadthFirst

} // class DirectedGraph
```

Uppg 6: Givet modulerna för stackar och köer, skall du göra en funktion `reverseQueue` som vänder på en kö med hjälp av en stack.

Tips: Det blir nog lättare om du definierar två hjälpfunktioner `queueToStack` och `stackToQueue`, antingen på huvudnivån eller i en `where`-sats.

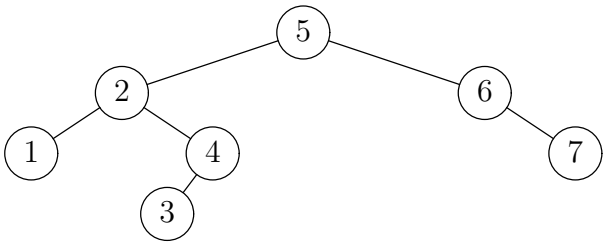
Obs: du skall *inte* göra detta inuti någon av modulerna !!

(6 p)

```
module Queue (
  Queue,      -- type of queues
  isEmpty,    -- Queue a -> Bool
  emptyQueue, -- Queue a
  enqueue,    -- a -> Queue a -> Queue a
  dequeue,    -- Queue a -> Queue a
  front       -- Queue a -> a
) where

module Stack (
  Stack,      -- type of stacks
  emptyStack, -- Stack a
  isEmpty,    -- Stack a -> Bool
  push,       --  a -> Stack a -> Stack a
  pop,        -- Stack a -> Stack a
  top         -- Stack a -> a
) where
```

Uppg 7: Givet ett binärt träd, **obs:** detta är *inte* ett *binärt sökträd*, vill du traversera trädet i **postorder**. Till exempel, för trädet:



skall elementen ges i följande ordning: 1, 3, 4, 2, 7, 6 och 5

Du skall deklarera en iterator för trädet som ger elementen i postorder. `remove` skall **inte** implementeras. För full poäng krävs att alla metoderna, inklusive konstruktorn, får högst ha en komplexitet av $\mathbf{O}(h)$, där h är trädets höjd.

Alla metoderna skall uppfylla beskrivningen som ges av API:n för `Iterator<E>` (se längst bak).

(12 p)

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

  protected TreeNode root;

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

   public Iterator<E> iterator() {
      return new BinTreeIterator();
   }

   /*** Här definierar du iteratorklassen  ***/
} // class BinTree
```