Institutionen för
Datavetenskap
Chalmers, Göteborgs universitet

VT10
DIT960
13/8-10

T E N T A M E N för

Datastrukturer DV, 7.5p, DIT960

DAG : 19 augusti 2010 Tid : 8.30-13.30 SAL : M

Ansvarig	:	Bror Bjerner, tel 772 10 29, 55 54 40
Resultat	:	senast den 25/8 -10 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, 5, 2, 3 och 4 i ett binärt sökträd. Du skall stoppa in dem i **exakt** denna ordning (dvs först 6, sedan 1, sedan 5, sedan 2 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 korrekt implementerat rödsvart träd är $\mathbf{O}(\,^2log\, n\,)$

b) Både i Java och Haskell kan köer implementeras effektivt med hjälp av en (enkellänkad) lista.

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

d) I en godtycklig oriktad graf gäller: Om man kan hitta ett *minimum spanning tree* enligt Kruskals algoritm, så får man exakt samma *minimum spanning tree* med Prims algoritm.

e) I en enkellänkad cirkulär lista tar det konstant tid att ta bort det första eller det sista elementet, om vi endast har en referens (pekare) till det sista elementet i listan.

f) Sammansmältningssorteringens (merge sort) komplexitet beror inte på hur elementen är ordnade i fältet.

(6 p)

Uppg 3: **a)**

20	10	50	30	60	40

Visa hur insättningssortering (insertion sort) fungerar genom att sortera fältet ovan. Rita upp fältet efter varje förändring i själva fältet. Rita även ut hjälpvariablernas värden efter fältet. (Ja, det blir lite 'tjatigt', men det är därför jag gjort fältet så litet och nästan sorterat)

(4 p)

b) Ange komplexiteten för bästa-, värsta- respektive genomsnitts-fallet. Ange också när vi har bästa fallet respektive värsta fallet. Slutligen, motivera genomsnittsfallet. **(4 p)**

Uppg 4: Heapar brukar lagras i fält. Avgör vilket av följande två fält, **a** eller **b**, som är en heap.

a:	1	3	7	4	5	8	6	9
index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

eller

b:	1	3	6	4	5	8	7	9
index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

a) Rita upp det fält, som är en heap, som ett binärt träd.

(1 p)

b) Motivera varför det andra fältet ej är en heap.

(1 p)

c) Stoppa in 2 i heapen och rita upp **fältet** efter instoppningen.

(2 p)

d) Ta sedan bort det minsta elementet och rita upp **fältet** efter borttagandet.

(2 p)

Del B

Datastrukturer på implementeringsnivå.

Uppg 5: Givet klasserna på nästa sida, skall du definiera en lokal iterator för grafen på angiven plats i koden. Iteratorn skall gå igenom alla noder som kan nås från en startnode enligt bredden först. Varje nod får bara förekomma en gång i uppräkningen.

Med bredden först menas här: först startnoden, därefter alla grannar till startnoden, du väljer själv i vilken ordning. Därefter alla grannars grannar, och så vidare. Notera att vikterna är irrelevanta.

För full poäng krävs att samtliga metoder och konstruktorn skall vara av så låg komplexitet som möjligt.

Vidare behöver du ej implementera `remove`. API:erna för metoderna för `Iterator<E>` finns längst bak i häftet.

(8 p)

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

public class DirectedGraph<E extends Edge>
       implements Iterable<Integer>              {

 private  List<E>[] neighbours;

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

 public void addEdge( E e ) {
    neighbours[e.from].add(e);
 } // addEdge

 //  Övriga metoder i grafen

 public Iterator<Integer> iterator() {
   return new DGIterator(0);
 } // default iterator

 public Iterator<Integer> iterator(int startnode) {
   return new DGIterator(startnode);
 } // iterator from startnode

 private class DGIterator
         implements Iterator<Integer> {

    /** HÄR SKRIVER DU DIN KOD  **/

 } // class DGIterator

} // class DirectedGraph
```

Uppg 6: Givet en modul för binära sökträd:

```
module BinSearchTree (
  BST,       -- type of binary search trees
  emptyTree, -- BST a
  isEmpty,   -- BST a -> Bool
  leftSub,   -- BST a -> BST a
  rightSub,  -- BST a -> BST a
  rootVal,   -- BST a -> a
  insert,    -- Ord a => a -> BST a -> BST a
  remove,    -- Ord a => a -> BST a -> BST a
  inorder,   -- BST a -> [a]
) where
```

skall du definiera två funktioner:

a) `height :: BST a -> Int`

som beräknar höjden av trädet. (Det tomma trädet skall ha höjden 0)

(2 p)

b) `preOrder :: BST a -> [a]`

som ger en lista av elementen i trädet i preorder. T.ex. för noderna i trädet i uppgift 7 blir listan:

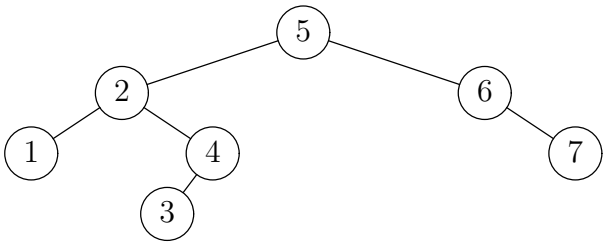
`[5,2,1,4,3,6,7]`

För poäng krävs att funktionen är av $\mathbf{O}(n)$

(6 p)

Obs: du skall *inte* göra dessa funktioner inuti modulen !!

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



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

Du skall deklarera en iterator för trädet som ger elementen i preorder. Konstruktorn och `remove` får högst ha en komplexitet av $\mathbf{O}(h)$, där h är trädets höjd. Övriga metoder skall ha en komplexitet av $\mathbf{O}(1)$.

Alla metoderna skall uppfylla beskrivningen som ges av API:n för `Iterator<E>` (se längst bak). Vidare måste `remove` implementeras så att ordningen för elementen ej förändras från den ursprungliga ordningen. Metoden `remove( E e )` är *inte* implementerad, dvs du måste implementera `remove` i iteratorn fullt ut. (Korrekt `remove` ger 8 p, övriga delar 6 p.)

(14 p)

```
public class BinTreeWithParent<E>
       implements Iterable<E>     {
  protected TreeNode<E> root;
  public static class TreeNode<E> {
    public E           element;
    public TreeNode<E> left, right, parent;
    public TreeNode( E           element,
                     TreeNode<E> left,
                     TreeNode<E> right
                     TreeNode<E> parent   ) {
           ... } // constructor TreeNode
   }  // class TreeNode
 public Iterator<E> iterator()
 { return new BinTreeIterator() }

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