TENTAMEN: Algoritmer och datastrukturer

Läs detta!

- *Uppgifterna är inte avsiktligt ordnade efter svårighetsgrad.*
- Börja varje uppgift på ett nytt blad.
- Skriv ditt idnummer på varje blad (så att vi inte slarvar bort dem).
- **Skriv rent dina svar. Oläsliga svar r ä t t a s e j!**
- Programkod skall skrivas i Java 5 eller senare version, vara indenterad och renskriven, och i övrigt vara utformad enligt de principer som lärts ut i kursen.
- Onödigt komplicerade lösningar ger poängavdrag.
- Programkod som finns i tentamenstesen behöver ej upprepas.
- Givna deklarationer, parameterlistor, etc. får ej ändras, såvida inte annat sägs i uppgiften.
- Läs igenom tentamenstesen och förbered ev. frågor.

I en uppgift som består av flera delar får du använda dig av funktioner klasser etc. från tidigare deluppgifter, även om du inte löst dessa.

Lycka till!

Uppgift 1

Välj **ett** svarsalternativ. Motivering krävs ej. Varje korrekt svar ger två poäng. Garderingar ger noll poäng.

1. När skiljer sig vanligen den genomsnittliga tidskomplexiteten (AC) från värstafallkomplexiteten (WC)?
   a. Vissa sorteringsalgoritmer
   b. Beräkning av maximala delsegmentsumman i ett fält
   c. Att vända innehållet i ett fält baklänges
   d. Sökning i en osorterad länkad lista

2. Om fältdubblering används vid implementering av en hashtabell och kvadratisk sondering används, vilket genomsnittligt minnesutnyttjande får man i tabellen?
   a. 10%
   b. 25%
   c. 37.5%
   d. 50%
   e. 75%

3. I kursen har vi arbetat med problemet att beräkna största rektangelsumman i en heltalsmatris. Vad kan vi säkert säga om tidskomplexiteten?
   a. Problemet är $\Omega(n^4)$
   b. Problemet är $\Omega(n^3)$
   c. Bästa kända algoritm är $O(n^2)$
   d. Bästa kända algoritm är $O(n^3)$

4. I vad bör man lagra elementen om man vill beräkna elementens median effektivt?
   a. Hashtabell
   b. Binär hög
   c. Länkad lista
   d. Binärt sökträd
   e. FIFO-kö
   f. Stack

5. Ange en minsta övre begränsning för kodavsnittets tidskomplexitet. Du kan anta att `TreeSet` är effektivt implementerad som ett balanserat sökträd.

```
TreeSet<Integer> t = new TreeSet<>();
// ... add m elements to t (omitted)
int[][]a = new int[n][n];
// ... add elements to a (omitted)
int count = 0;
for ( int i = 0; i < a.length; i++ )
    for ( int j = 0; j < a[0].length; j++ )
        if ( t.contains(a[i][j] )
            count++;
```

   a. $O(n^2)$
   b. $O(n^2 \cdot m)$
   c. $O(n^2 \cdot \log n)$
   d. $O(n^2 \cdot \log m)$
   e. $O(n^3)$

(10 p)

Uppgift 2

Följande typ kan användas för att representera noder i ett binärt träd.

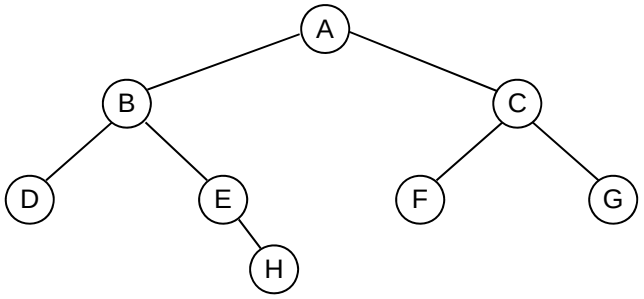
```
public class TreeNode {
    char element;
    TreeNode left,right;

    public boolean isLeaf() {
          return left == null && right == null;
    }
}
```

Skriv en funktion som returnerar en lista av alla löv i ett träd.

```
public static List<Character> getLeaves(TreeNode t)
```

Listan skall innehålla elementen som finns i löven.

Exempel: För trädet nedan skall en lista med D, H, F och G returneras.



(10 p)

Uppgift 3

a) Vid genomlöpning av ett binärt träd besöktes noderna inorder: B, D, A, G, E, J, H, K, C, I, F och preorder: A, B, D, C, E, G, H, J, K, F, I. Rita trädet!

(2 p)

b) Rita ett valfritt AVL-träd med höjden 4 och minimalt antal noder.

(1 p)

Uppgift 4

a) Ange villkor för att ett träd skall vara en binär hög.

(2 p)

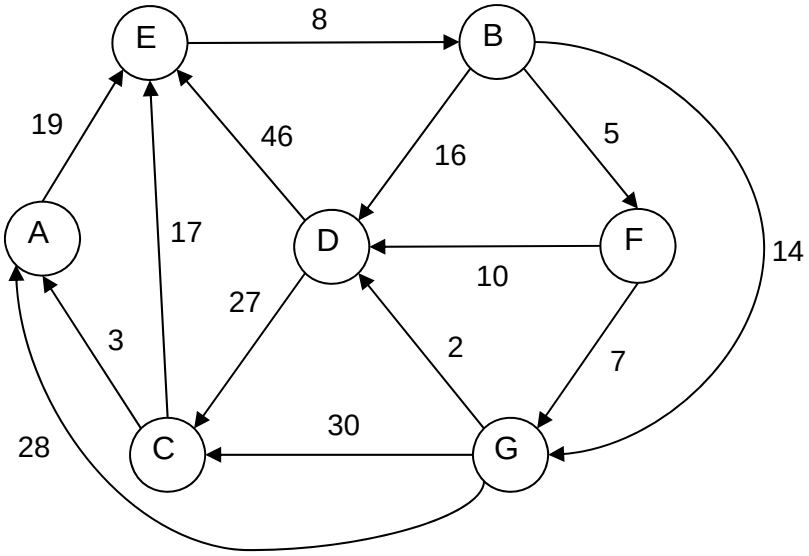
b) Utför `deleteMin();insert(11);deleteMin();` på den binära högen nedan och rita resultatet som ett träd.

-∞	10	13	12	18	16	14	15	28	19
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

(3 p)

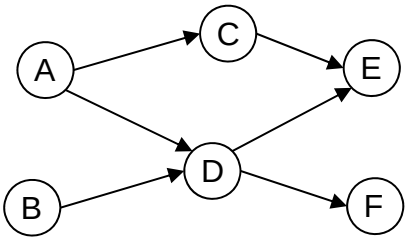
Uppgift 5

a) Beräkna de kortaste viktade avstånden från nod B till alla övriga noder med Dijkstras algoritm i grafen nedan. Ange i vilken ordning noderna besöks. Ange för varje nod samtliga approximationer av det kortaste avståndet till noden som beräknas i algoritmen.



(6 p)

b) Ange alla möjliga topologiska ordningar av noderna i grafen:



(6 p)

Uppgift 6

I hashtabellen nedan tillämpas kvadratisk sondering för kollisionshantering.
Hashfunktionen är *hash*(*x*) = *x* % *M*, där *M* är tabellstorleken.

0	
1	
2	
3	
4	
5	
6	

a) Utför insättningssekvensen

```
insert(18);insert(40);insert(32);insert(25);
```

i tabellen och visa hur resultatet blir.

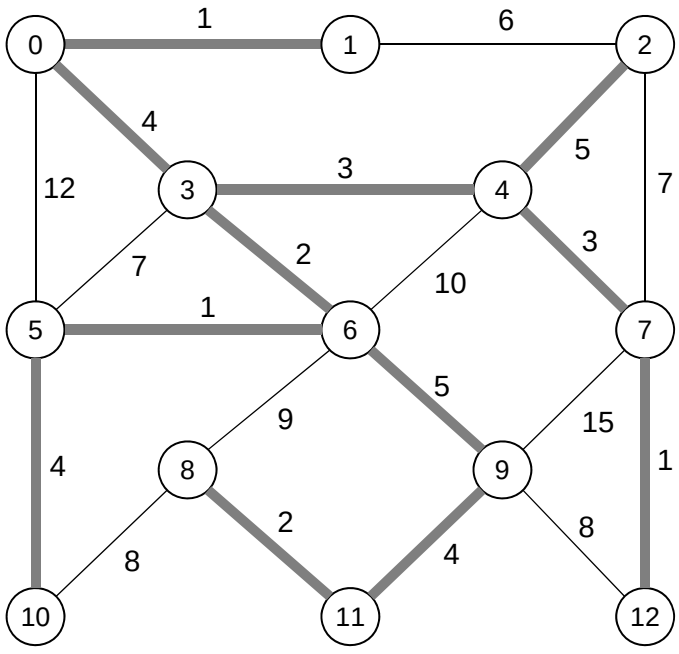
(4 p)

b) Vad händer om du försöker utföra `insert(4)` efter att de fyra elementen har satts in i a? Varför fungerar det inte? Hasha om elementen till en större tabell av lämplig storlek.

(4 p)

Uppgift 7

Ett uppspännande träd i en sammanhängande oriktad graf är en delmängd av grafens bågar som förbinder alla noder med varandra. I det här exemplet är ett sådant träd ritat med fet linje:



Det kan finnas flera möjliga sådana träd. I denna uppgift skall vi inrikta oss på uppspännande träd med minimal sammanlagd bågkostnad. Trädet ovan har just denna egenskap. Grafbågar representeras med typerna:

```
public class Edge implements Comparable<Edge> {
    public double cost;
    public int src,dest;

    public Edge(int src,int dest,double cost) {
        this.src = src;
        this.dest = dest;
        this.cost = cost;
    }
    public int compareTo(Edge other) {
        return cost < other.cost ? -1 :
               cost > other.cost ? 1 :
               0;
    }
}
public class EdgeComparator implements Comparator<Edge> {
    public int compare(Edge lhs,Edge rhs) {
        return lhs.compareTo(rhs);
    }
}
```

Bågarnas riktning spelar ingen roll i uppgiften, det viktiga är vilka två noder en båge förbinder. Implementera metoden `minimalTree`. Metoden skall ha signaturen:

```
public static List<Edge> minimalTree(List<Edge> edges)
```

inparametern innehåller grafens bågar och returvärdet skall vara en lista med bågar som bildar ett minimalt uppspännande träd. Använd lämpliga datastrukturer i lösningen.
Några tips: Välj bågarna i storleksordning när du bygger trädet. Du kommer att behöva räkna ut antalet noder i grafen. Gör det i en separat privat hjälpmetod och använd även där en lämplig datastruktur.

(12 p)