TENTAMEN: Algoritmer och datastrukturer

Läs detta!

- *Uppgifterna är inte avsiktligt ordnade efter svårighetsgrad.*
- Börja varje uppgift på ett nytt blad.
- Skriv din tentamenskod på varje blad (så att vi inte slarvar bort dem).
- **Skriv rent dina svar. Oläsliga svar r ä t t a s e j!**
- Programmen skall skrivas i Java 5 eller senare version, vara indenterade och renskrivna, och i övrigt vara utformade 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!

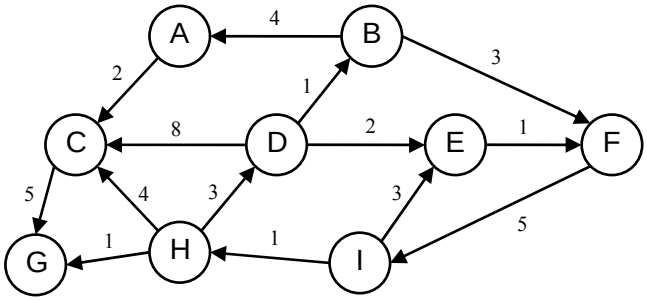
Uppgift 1

Välj **ett** svarsalternativ på frågorna 2, 3 och 5. Motivering krävs ej. Varje korrekt svar ger två poäng. Garderingar ger noll poäng.

1. Vad är tidskomplexiteten för att sätta in ett element i en fältbaserad datastruktur som utnyttjar kostnadsamortering genom fältdubblering?

   a. $O(1)$ i genomsnitt och $O(N)$ i värsta fall
   b. $O(N)$ i genomsnitt och $O(N)$ i värsta fall
   c. $O(1)$ i genomsnitt och $O(1)$ i värsta fall
   d. $O(N)$ i genomsnitt och $O(1)$ i värsta fall

2. I vilken ordning besöker Dijkstras algoritm noderna om sökningen startar i D?



   a. DBECAFGIH
   b. DBEFACIHG
   c. DBAFIECHG
   d. DBEFAIHCG

3. Hur många inversioner innehåller ett genomsnittligt heltalsfält?
   a. $O(\log n)$
   b. $\Omega(n^2)$
   c. $O(n \log n)$
   d. $O(n)$
   e. $\Omega(n^2 \log n)$

4. Antag att vi vill implementera datastrukturerna i tabellen nedan med generiska javaklasser. Vilka operationer krävs för att hantera elementen i respektive struktur? Markera med ett ”X” om operationen är nödvändig och med ett ”-” om den inte är det. För poäng krävs att alla tabellrutor fyllts i korrekt.

	Operation		
Datastruktur	**`compareTo`**	**`hashCode`**	**`equals`**
Binärt sökträd			
Hashtabell			
Binär hög			
Stack			
FIFO-kö			

forts. på nästa sida ->

5. Antag att **c** är en positiv konstant. Ange en minsta övre begränsning bland alternativen a-f för kodavsnittets tidskomplexitet.

```
int sum = 0;
for ( int i = 0; i < n; i++ )
    for ( int j = 0; j < n; j++ )
        for ( int y = -1; y <= 1; y++ )
            for ( int x = -1; x <= 1; x++ )
                for ( int k = 0; k < n; k++ ) {
                    if ( k > c )
                        break;
                    sum++;
                }
```

a. $O(n^2)$
b. $O(n^3)$
c. $O(6n^3)$
d. $O(6\mathbf{c}n^2)$ där c är en positiv konstant
e. $O(n^4)$
f. $O(n^5)$

(10 p)

Uppgift 2

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

```
public class TreeNode {
    int element,height,size;
    TreeNode left,right;

    public TreeNode(int element,TreeNode left,TreeNode right) {
        this.element = element;
        this.left = left;
        this.right = right;
        height = 1 + Math.max(height(left),height(right));
        size = 1 + size(left) + size(right);
    }
    public boolean isLeaf() {
       return left == null && right == null;
    }
    public static int height(TreeNode t) {
       return t == null ? 0 : t.height;
    }
    public static int size(TreeNode t) {
       return t == null ? 0 : t.size;
    }
}
```

a) Skriv en rekursiv funktion som avgör om ett träd är fullt. Ett tomt träd är per definition fullt.

```
public static boolean isFull(TreeNode t)
```

Lösningen får ej baseras på numeriska samband mellan trädets höjd och antal noder.

(4 p)

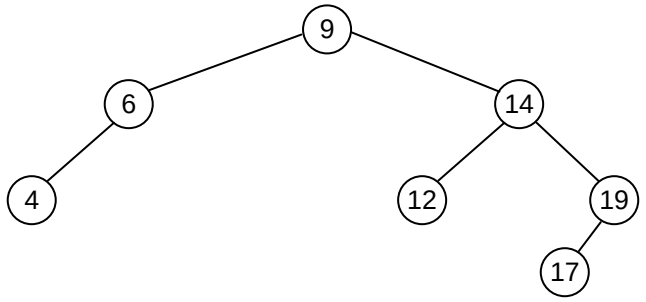
b) Skriv en rekursiv funktion som avgör om ett träd är komplett, d.v.s. att det uppfyller strukturvillkoret för att vara en binär hög. Ett tomt träd är per definition komplett.

```
public static boolean isComplete(TreeNode t)
```

(6 p)

Uppgift 3

Ange vilka insättningsföljder av 5, 8 och 23 som kan utföras i trädet nedan utan att bryta mot AVL-villkoret. Rita det resulterande trädet. Några insättningsfölder kräver en AVL-rotation. Vilka följder, och hur ser trädet ut efter dessa?



Du skall alltså rita två olika träd.

(6 p)

Uppgift 4

Givet följande datatyp för listnoder

```
public class ListNode {
    int element;
    ListNode next;

    public ListNode(int x,ListNode n) {
        element = x;
        next = n;
    }
}
```

konstruera en metod som sätter in ett element i en lista där elementen är ordnade i växande följd.

```
public static ListNode insert(int x,ListNode l)
```

Efter insättningen skall förstås listan fortfarande vara ordnad. Lös problemet iterativt med en loop, eller rekursivt, vilket du vill. I en iterativ lösning kan du anta att listans första nod är ett listhuvud utan datainnehåll. I en rekursiv lösning skall listan ej ha någon sådan huvudnod. Efter insättningen skall en referens till listans första nod (eller huvudnoden) returneras.

(6 p)

Uppgift 5

Dimensionera en hashtabell med det minsta antalet platser som lämpar sig för lagring av sex element då kvadratisk sondering används som kollisionshanteringsmetod. Visa med en figur hur en initialt tom sådan tabell ser ut efter sekvensen

```
add(20); add(29); add(3); remove(3); add(51); remove(20); add(4); add(42);
```

Hashfunktionen definieras som hash(x) = x mod M, där M är tabellstorleken.

(5 p)

Uppgift 6

a) Rita en graf med fem noder märkta A-E sådan att BACDE, BADCE, ABCDE, ABDCE, ACBDE är de enda möjliga topologiska ordningarna av noderna i grafen.

(5 p)

b) Studera följande kod för representation av oriktade grafer

```
public class Edge implements Comparable<Edge> {
    int first,second;
    Integer cost;
    public Edge(int first,int second,int cost) {
        this.first = first;
        this.second = second;
        this.cost = cost;
    }
    public int compareTo(Edge other) {
        return cost.compareTo(other.cost);
    }
}
public class Graph {
    public void addEdge(Edge e); // add e to this graph
       ...
}
...
public static void f(Graph g,int n,List<Edge> edges) {
    PriorityQueue<Edge> pq = new PriorityQueue<Edge>(edges);
    DisjointSets ds = new DisjointSets(n);
    int unions = 0;
    while ( unions < n - 1 ) {
        Edge e = pq.poll();
        int s1 = ds.find(e.first), s2 = ds.find(e.second);
        if ( s1 != s2 ) {
            ds.union(s1,s2);
            unions++;
            g.addEdge(e);
        }
    }
}
```

Rita grafen som byggs när nedanstående kod exekveras

```
List<Edge> edges =
     Arrays.asList(
         new Edge[]{
             new Edge(0,1,4),
             new Edge(0,2,5),
             new Edge(1,5,3),
             new Edge(1,3,1),
             new Edge(2,3,6),
             new Edge(3,4,2),
             new Edge(4,5,4),
             new Edge(2,6,5),
             new Edge(2,7,4),
             new Edge(3,7,4),
             new Edge(7,8,1),
             new Edge(4,8,3),
             new Edge(5,8,5),
             new Edge(6,7,3),
             new Edge(6,9,2),
             new Edge(7,9,4)
         });
f(new Graph(),10,edges);
```

(6 p)

Uppgift 7

En binär hög är som bekant ett komplett träd som vanligen lagras i ett fält. Denna uppgift går ut på att konstruera två metoder som kan konvertera mellan fält- och trädrepresentation. Vi använder samma trädtyp som i uppgift 2. Det är även tillåtet att utnyttja funktioner som konstruerades i uppgift 2.

a) Skriv en funktion som tar en binär hög representerad som ett fält som argument och returnerar en trädrepresentation av högen.

```
public static TreeNode heapAsTree(int[] a)
```

Tips. Låt metoden anropa en rekursiv hjälpmetod som tar fältet och index för roten till ett delträd som parametrar. Roten till hela trädet finns som vanligt i position 1.

(4 p)

b) Skriv en funktion som tar en binär hög representerad som ett träd som argument och returnerar en fältrepresentation av högen.

```
public static int[] treeAsHeap(TreeNode t)
```

Använd en lämplig datastruktur i algoritmen.

(8 p)