TENTAMEN: Algoritmer och datastrukturer

Läs detta!

- *Uppgifterna är inte avsiktligt ordnade efter svårighetsgrad.*
- Börja varje uppgift på ett nytt blad.
- Skriv ditt namn och personnummer 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, vara indenterade och kommenterade, 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 varje fråga. Motivering krävs ej. Varje korrekt svar ger två poäng. Garderingar ger noll poäng.

1. Vilken av följande metoder för insättning av ett nytt element först i en länkad lista utan huvud är korrekt? Klassen `ListNode` definieras

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

a.
```
void insert(int x,ListNode l) {
    ListNode tmp = new ListNode();
    tmp.element = x;
    tmp.next = l;
    l = tmp;
}
```

b.
```
ListNode insert(int x,ListNode l) {
    ListNode tmp = new ListNode();
    tmp.element = x;
    tmp.next = l;
    return l;
}
```

c.
```
ListNode insert(int x,ListNode l) {
    ListNode tmp = new ListNode();
    tmp.element = x;
    tmp.next = l;
    return tmp;
}
```

d.
```
ListNode insert(int x,ListNode l) {
    ListNode tmp = new ListNode();
    tmp.element = x;
    tmp.next = l.next;
    l.next = tmp;
    return l;
}
```

2. Antag att man vill konstruera ett program som rankar löparnas sluttider i Göteborgsvarvet. På en resultattavla visas de tio bästa placeringarna, ordnade efter stigande sluttider. Resultatlistan uppdateras vid varje förändring bland topp tio. Vilken datastruktur lämpar sig bäst för att hålla reda på löparnas tider och effektivt leverera informationen som behövs för att uppdatera resultattavlan?
   a. Hashtabell
   b. Binärt sökträd
   c. Stack
   d. Prioritetskö
   e. Map
   f. FIFO-kö

3. Ineffektiva rekursiva algoritmer med många överlappande fall kan ofta göras mer effektiva genom att utnyttja
   a. en snål algoritm
   b. backtracking
   c. divide and conquer
   d. dynamisk programmering

4. Ett av sambanden 1-3 beskriver tidskomplexiteten hos f.

1.
$$\begin{cases} T(0) = 0 \\ T(N) = T(N-1) + N \end{cases}$$

2.
$$\begin{cases} T(0) = 0 \\ T(N) = T(N/2) + N \end{cases}$$

3.
$$\begin{cases} T(0) = 0 \\ T(N) = T(N-1) + \log N \end{cases}$$

```
int f(int n) {
    if ( n == 0 )
        return 0;
    else {
        int s = f(n-1);
        for ( int i = n; i > 0; i /= 2 )
            s += 1;

        return s;
    }
}
```

Vilket av a-d är en minsta övre begränsning för lösningen till sambandet i fråga?
   a. $O(\log N)$
   b. $O(N)$
   c. $O(N \cdot \log N)$
   d. $O(N^2)$

5. Alla sorteringsalgoritmer som baseras på successiva byten av närliggande element har generellt tidskomplexiteten
   a. $O(N \cdot \log N)$ i genomsnitt
   b. $O(N \cdot \log N)$ i värsta fall
   c. $\Omega(N^2)$ i bästa fall
   d. $\Omega(N^2)$ i genomsnitt
   e. $O(N^2)$ i värsta fall

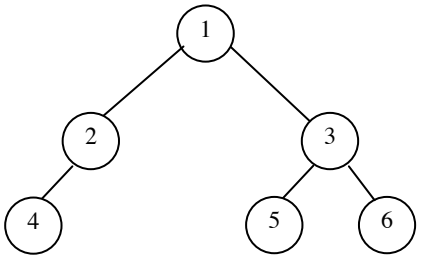
(10 p)

Uppgift 2

Konstruera en rekursiv funktion

```
public static ListNode getLevel(TreeNode t, int i)
```

som returnerar en lista med alla element som finns på en viss nivå i ett binärt träd. Exempel: Om `t` är trädet

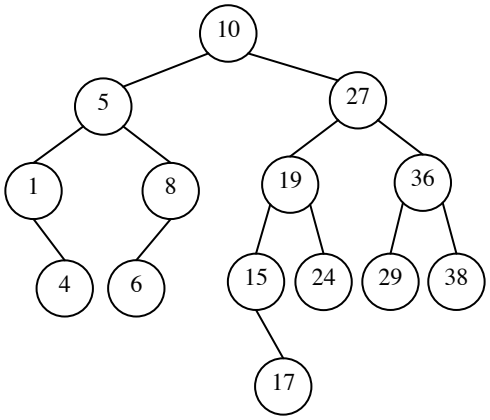


så skall anropen `getLevel(t,0)`, `getLevel(t,1)`, `getLevel(t,2)` returnera listorna `[1]`, `[2,3]`, respektive `[4,5,6]`. För `i > 2` returnerar `getLevel(t,i)` **`null`**. Klassen `ListNode` finns i uppgift 1.1.

(10 p)

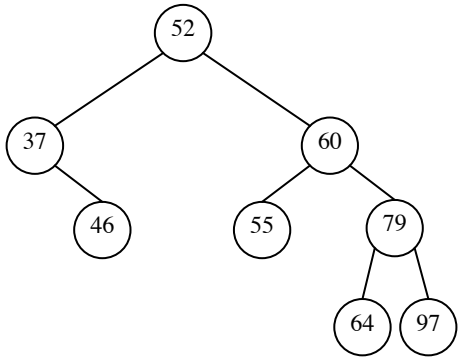
Uppgift 3

a) Visa hur det binära sökträdet nedan ser ut efter operationen `delete(10)`.



(2 p)

b) Visa hur det binära sökträdet nedan ser ut efter operationen `insert(82)`. Efter insättningen skall trädet uppfylla AVL-villkoret.



(3 p)

Uppgift 4

I en tom hashtabell med sju platser gjordes insättningssekvensen `insert(31); insert(18); insert(49);` vilket resulterade i tabellen

0	49
1	
2	
3	31
4	18
5	
6	

Hashfunktionen är här för enkelhets skull $hash(x) = x \; mod$ M där M är antalet platser i tabellen. Kvadratisk sondering (probing) används.

a) Är det lämpligt att sätta in ytterligare två element i tabellen ovan, utan andra förändringar? Motivera svaret!

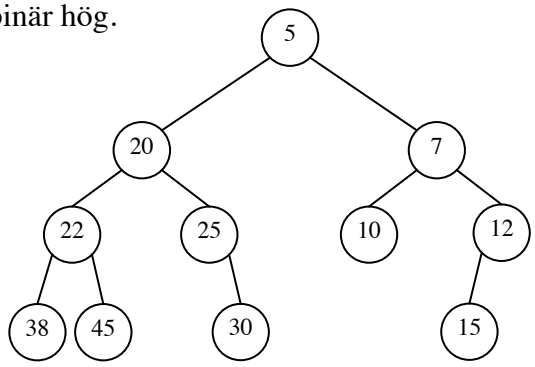
(2 p)

b) Välj minsta lämpliga tabellstorlek som är minst dubbelt så stor som ovan. Visa sedan hur denna tabell ser ut efter sekvensen `insert(31); insert(18); insert(49); insert(48);` Använd kvadratisk sondering (probing).

(4 p)

Uppgift 5

a) Är trädet nedan en binär hög? Om inte, förklara varför och flytta i så fall element på lämpligt sätt så att det blir en binär hög.



(3 p)

b) Visa hur fältrepresentationen av högen i a ser ut.

(2 p)

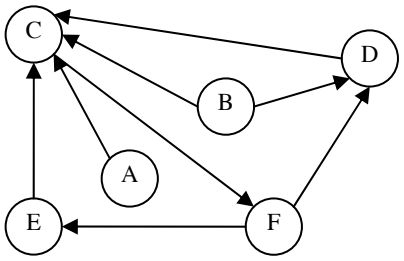
Uppgift 6

För att kunna ordna noderna i grafen nedan topologiskt krävs att en ny båge läggs till eller att en befintlig tas bort. Gör detta och ange sedan alla möjliga topologiska ordningar av noderna.

Ledning: Inför förkortningen X{$v_1 \ldots v_n$}Y för mängden av alla sekvenser som börjar med X, därefter en permutation av $v_1 \ldots v_n$ och sist Y.

Exempel: a{xyz}bc, motsvarar de sex olika sekvenserna axyzbc, axzybc, ayxzbc, ayzxbc, azxybc samt azyxbc eftersom de tre elementen xyz kan permuteras på 3! = 6 olika sätt.

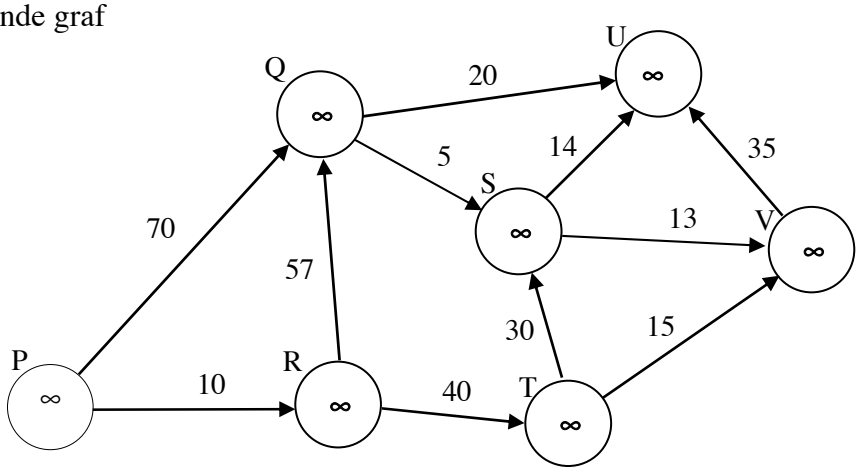
Utnyttja skrivsättet ovan som en förkortning för att beskriva de topologiska ordningarna.



(7 p)

Uppgift 7

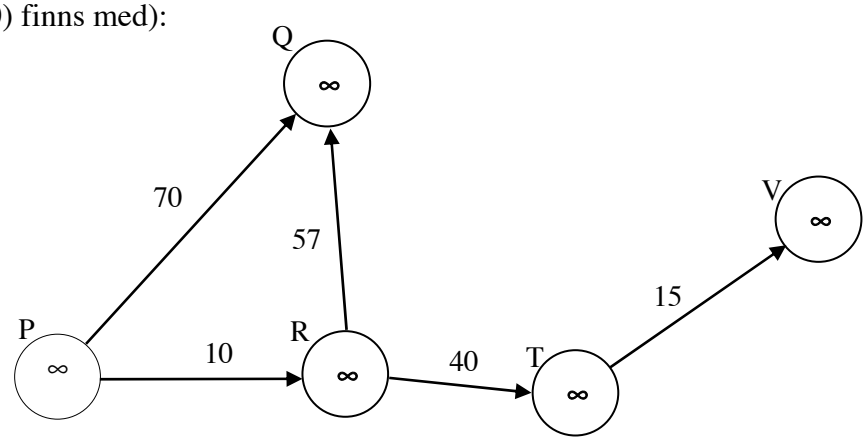
I denna uppgift används grafklassen i kursboken. Den finns som bilaga längst bak i tesen.
Betrakta följande graf



a) Lägg till metoden

```
public Graph computeSubGraph(String startNode,double maxDist)
```

i grafklassen. Metoden skall returnera en ny graf som innehåller alla noder i aktuell graf som kan nås inom avstånd `maxDist` från `startNode`. Alla bågar i aktuell graf som har både start- och slutnod som också finns i subgrafen skall finnas med i subgrafen. Exempel: Om startnoden är P och maxavståndet 67 så skall metoden returnera delgrafen (observera att bågen (P,Q,70) finns med):



Ledning: Metoden `dijkstra` i grafklassen får utnyttjas i lösningen.

(7 p)

b) Lägg till metoden

```
public List<String> nodesAtUWD(String startNode,int d)
```

i grafklassen. Metoden skall returnera en lista med alla noder i aktuell graf som kan nås på *oviktat* avstånd (exakt) `d` från `startNode`. Exempel: Om startnoden är P och avståndet 0, 1, 2, 3, respektive 4 eller större, så skall metoden returnera listan [P], [R,Q], [S,T,U], [V], respektive []. (listelementens inbördes ordning spelar ingen roll).
Ledning: Klassen `Pair` sist i bilagan är användbar i lösningen.

(10 p)

BILAGA

```
// Represents an edge in the graph.
class Edge {
    public Vertex     dest;   // Second vertex in Edge
    public double     cost;   // Edge cost

    public Edge( Vertex d, double c ) {
        dest = d;
        cost = c;
    }
}

// Represents an entry in the priority queue for Dijkstra's algorithm.
class Path implements Comparable<Path> {
    public Vertex     dest;   // w
    public double     cost;   // d(w)

    public Path( Vertex d, double c ) {
        dest = d;
        cost = c;
    }

    public int compareTo( Path rhs ) {
        double otherCost = rhs.cost;
        return cost < otherCost ? -1 : cost > otherCost ? 1 : 0;
    }
}

// Represents a vertex in the graph.
class Vertex {
    public String     name;   // Vertex name
    public List<Edge> adj;    // Adjacent vertices
    public double     dist;   // Cost
    public Vertex     prev;   // Previous vertex on shortest path
    public int        scratch;// Extra variable used in algorithm

    public Vertex( String nm )
      { name = nm; adj = new LinkedList<Edge>( ); reset( ); }

    public void reset( )
      { dist = Graph.INFINITY; prev = null;  scratch = 0; }
}

// Graph class: evaluate shortest paths.
//
// CONSTRUCTION: with no parameters.
//
// ******************PUBLIC OPERATIONS*********************************
// void addEdge( String v, String w, double cvw )
//                              --> Add additional edge
// void dijkstra( String s )    --> Single-source weighted
// Graph computeSubGraph( String s, double d) --> Sub graph within d
// List<String> nodesAtUWD(String s, int d)   --> Nodes at distance d
// ************************************************************************
```

v.g.v. -->

```
public class Graph {
    public static final double INFINITY = Double.MAX_VALUE;
    private Map<String,Vertex> vertexMap =
         new HashMap<String,Vertex>( );

    /**
     * Add a new edge to the graph.
     */
    public void addEdge( String sourceName, String destName,
           double cost )
    {
        Vertex v = getVertex( sourceName );
        Vertex w = getVertex( destName );
        v.adj.add( new Edge( w, cost ) );
    }

    /**
     * If vertexName is not present, add it to vertexMap.
     * In either case, return the Vertex.
     */
    private Vertex getVertex( String vertexName ) {
        Vertex v = vertexMap.get( vertexName );
        if( v == null )
            v = new Vertex( vertexName );
            vertexMap.put( vertexName, v );
        }
        return v;
    }

    /**
     * Initializes the vertex output info prior to running
     * any shortest path algorithm.
     */
    private void clearAll( ) {
        for( Vertex v : vertexMap.values( ) )
            v.reset( );
    }

    public void dijkstra( String startName ){ ... }

    // uppgift 7a
    // Compute the subgraph of this graph where all nodes are
    // reacheable from startNode within at most maxDist distance.
    // Nodes at longer distance are omitted.
    public Graph computeSubGraph(String startNode,double maxDist) ...

    // uppgift 7b
    // Compute a list of all nodes at the exact
    // unweighted distance d from startNode
    public List<String> nodesAtUWD(String startNode,int d) ...
}
```