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!**
- 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. Om en FIFO-kö implementeras med fältdubbleringstekniken som beskrivs i kursboken så får operationen enqueue som adderar ett element till kön tidskomplexiteten
   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 värsta fall

2. Om man skall asfaltera delar av ett vägnät mellan ett antal orter så att alla orter förbinds med nyasfalterad väg, till minimal kilometerkostnad, så behövs, förutom asfalt, följande datastrukturer för att beräkna det optimala subvägnätet:
   a. FIFO-kö och binärt sökträd
   b. Prioritetskö och DisjointSets
   c. HashMap
   d. Stack och HashSet

3. Vilken/vilka av nedanstående sorteringsalgoritm(er) bygger på divide & conquer?
   a. endast shellsort
   b. endast mergesort
   c. endast quicksort
   d. endast insertionsort
   e. quicksort och mergesort
   f. shellsort och quicksort
   g. mergesort och shellsort

4. Antag att man har ett visst problem och vill veta hur snabbt det kan lösas.
   Då bör man
   a. konstruera en algoritm, hitta en lämplig funktion f, och visa att $O(f(n))$ är en minsta övre begränsning för algoritmens tidskomplexitet.
   b. konstruera en algoritm, hitta en lämplig funktion f, och visa att $\Omega(f(n))$ är en största undre begränsning för algoritmens tidskomplexitet.
   c. hitta en lämplig funktion f och visa att problemet för maximalt ogynnsamma indata kräver högst $O(f(n))$ beräkningssteg.
   d. hitta en lämplig funktion f och visa att problemet för genomsnittliga indata kräver minst $\Omega(f(n))$ beräkningssteg.

5. Antag att vi har följande definitioner: s är en sekvens av noder, t är ett binärt träd, reverse(s) ger omvändningen av s, mirror(t) ger trädet som är speglingen av t (på samma sätt som i laboration 5), postorder(t) ger sekvensen av t:s noder i postorder ordning, samt preorder(t) ger sekvensen av t:s noder i preorder ordning.
   Vilket påstående är sant?
   a. reverse(postorder(t)) = preorder(mirror(t))
   b. postorder(t) = reverse(preorder(t))
   c. preorder(t) = postorder(mirror(t))

(10 p)

Uppgift 2

I många ordbehandlingsprogram finns möjlighet att automatiskt producera en innehållsförteckning över alla sektioner i ett dokument. Alla rubriker på olika nivåer i dokumentet kommer med i innehållsförteckningen. Varje rad börjar med rubrikens sektionsnummer följt av själva rubriktexten. Sektioner kan vara hierarkiskt uppbyggda med undersektioner, som i sin tur kan innehålla undersektioner o.s.v. Följande klass representerar sektionshierarkin i en innehållsförteckning:

```
public class Section {
    private String title;
    private List<Section> subsections;

    public Section(String title) {
        this.title = title;
        subsections = new ArrayList<Section>();
    }

    public void add(Section m) {
        subsections.add(m);
    }

    public void print() {
        // implement this!
    }
}
```

Implementera metoden `print`. Algoritmen skall vara rekursiv. Följande exempel visar hur utskriften skall se ut. *Tips*: Om det underlättar får du definiera en privat rekursiv hjälpmetod som anropas av `print` på lämpligt sätt.

```
Section alg = new Section("Algorithms");
alg.add(new Section("Recursive algorithms"));
alg.add(new Section("Sorting algorithms"));
alg.add(new Section("Correctness of algorithms"));

Section c = new Section("Complexity of algorithms");
c.add(new Section("Time Complexity"));
c.add(new Section("Space Complexity"));
c.add(new Section("Worst Case behaviour"));
c.add(new Section("Average Case behaviour"));
alg.add(c);

Section ds = new Section("Data Structures");
ds.add(new Section("Abstract data types"));

Section lists = new Section("Lists");
lists.add(new Section("Array based representation"));
lists.add(new Section("Pointer based representation"));
ds.add(lists);

Section trees = new Section("Trees");
trees.add(new Section("General trees"));
Section btrees = new Section("Binary trees");
btrees.add(new Section("Expression trees"));
btrees.add(new Section("Huffman coding trees"));
btrees.add(new Section("Binary search trees"));
trees.add(btrees);
ds.add(trees);

Section algDS = new Section("Algorithms and Data Structures");
algDS.add(alg);
algDS.add(ds);

algDS.print();
```

utskrift

```
1 Algorithms and Data Structures
1.1 Algorithms
1.1.1 Recursive algorithms
1.1.2 Sorting algorithms
1.1.3 Correctness of algorithms
1.1.4 Complexity of algorithms
1.1.4.1 Time Complexity
1.1.4.2 Space Complexity
1.1.4.3 Worst Case behaviour
1.1.4.4 Average Case behaviour
1.2 Data Structures
1.2.1 Abstract data types
1.2.2 Lists
1.2.2.1 Array based representation
1.2.2.2 Pointer based representation
1.2.3 Trees
1.2.3.1 General trees
1.2.3.2 Binary trees
1.2.3.2.1 Expression trees
1.2.3.2.2 Huffman coding trees
1.2.3.2.3 Binary search trees
```

(12 p)

Uppgift 3

En av insättningssekvenserna

10, 20, 6, 15, 4, 5, 30
5, 3, 10, 8, 12, 7
10, 15, 12, 17, 22, 5, 6
10, 5, 7, 3, 15, 20, 1, 18

ger, om den utförs på ett tomt binärt sökträd, ett träd som uppfyller AVL-villkoret.
Rita trädet!

(4 p)

Uppgift 4

Antag att vi i en viss hashtabell använder kvadratisk sondering vid insättning av nya element.
Vad är det för fel på följande två tabeller?

a) (1 p)

0	
1	7
2	33
3	
4	1
5	139
6	
7	

b) (1 p)

0	28
1	
2	4
3	
4	19
5	
6	34

c) I en hashtabell med åtta platser används linjär sondering vid insättning. Visa hur tabellen ser ut efter sekvensen add(26); add(12); add(34); remove(26); add(2); Hashfunktionen definieras *hash*(*x*) = *x mod M,* där *M* är antalet platser i tabellen.

(4 p)

Uppgift 5

Fältet nedan är en linjär representation av ett komplett träd

	14	8	12	23	10	9	13	20	19	11				
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*

a) Rita trädet

(1 p)

b) Rita trädet som det ser ut efter operationen `buildHeap`.

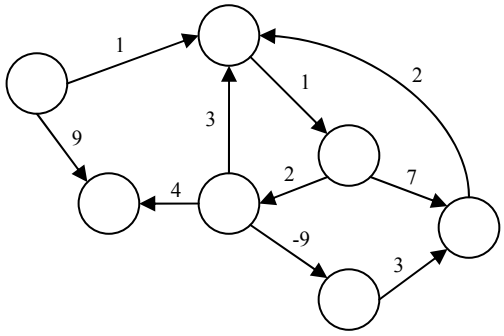
(2 p)

c) Ange ordokomplexiteterna i genomsnitt resp. i värsta fall för operationerna `insert` och `buildHeap`.

(4 p)

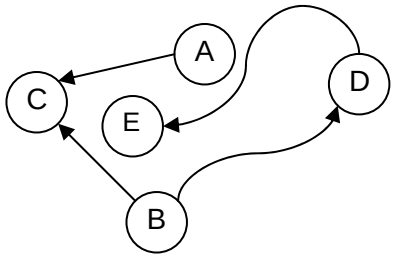
Uppgift 6

a) Dijkstras algoritm kan inte beräkna de kortaste viktade vägarna i följande graf. Varför inte? Ge ett exempel som illustrerar problemet!



(2 p)

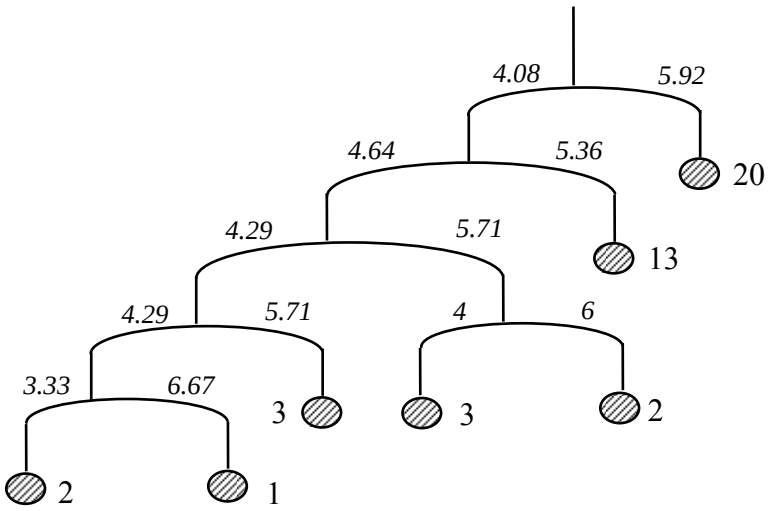
b) Lista alla möjliga topologiska ordningar av noderna i grafen



(6 p)

Uppgift 7

Denna uppgift går ut på att bygga en balanserad mobil från en lista av vikter (gram). Om indata är t.ex. listan är [1,13,2,3,20,2,3] så skall följande mobil byggas (så när som på speglingar av delmobiler).



Figuren är inte skalenlig m.a.p. de horisontella pinnarnas inbördes längder. Mobilen skall konstrueras så att de lättaste vikterna hamnar långt ner, och de tyngsta långt upp. Genom att bygga så får den ett mer välbalanserat utseende genom att de lättare vikterna samlas i delmobiler som kan vägas upp av enstaka tyngre vikter, vilket framgår av exemplet ovan. För att förenkla lite skall alla pinnar ha samma längd (se konstant i koden). I verkligheten skulle det begränsa rotationsfriheten, men vi bortser från det nu. Dessutom antar vi att varje pinne väger exakt 1 gram, men de vertikala snörenas vikt bortser vi från. Mobilen ovan väger alltså 50 gram. Följande ofullständiga klass är given

```
public class Mobile {
    public static int ROD_LENGTH = 10;
    public static int ROD_WEIGHT = 1;
    private enum MobileType { SIMPLE, COMPOSITE }
    private MobileType type;
    private float weight;                    // Simple
    private float leftLength, rightLength;  // Composite
    private Mobile left, right;

    // Simple case
    public Mobile( float weight ) {
        type = MobileType.SIMPLE;
        this.weight = weight;
        left = right = null;
    }

    // Composite case.
    public Mobile(Mobile left, Mobile right) { // Implement! }

    // Determines if this mobile is simple.
    private boolean isSimple() { return type == MobileType.SIMPLE; }

    // Returns -1 if the weight of this mobile is less than that of the
    // given mobile, 1 if its is greater, and 0 ow.
    public int compareTo(Mobile m) { ... }

    // Returns the total mass of the mobile.
```

```
    public float getWeight() { return weight; }

    // Returns a balanced mobile constructed from the given weights.
    public static Mobile build(List<Integer> weights) { // Implement! }

}
```

Notera att variabeln `weight` skall ha korrekt värde i alla noder, både löv och sammansatta.

a) Implementera den andra konstruktorn som bygger en sammansatt mobil givet två delmobiler. Konstruktorn skall själv räkna ut lämpliga längder på den sammanbindande pinnens vänstra och högra del.

(4 p)

b) Implementera metoden `build` enligt beskrivningen ovan. Om listan är tom skall null returneras. Om den bara innehåller ett element returneras en enkel mobil. Använd lämplig datastruktur som hjälp i byggprocessen. Metoden behöver ej vara rekursiv.

(9 p)