Data och Informationsteknik / Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Erland Holmström

Göteborg 12 Mar 2018

Examination i

Datastrukturer och Algoritmer IT, TDA416

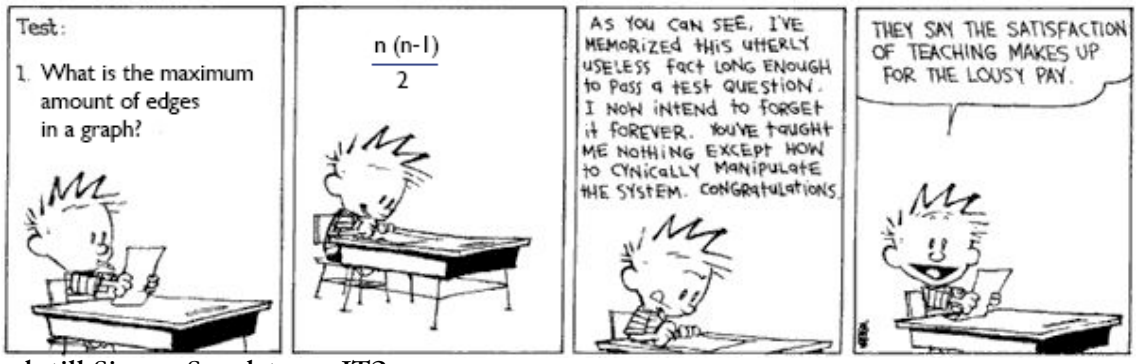
Dag: Lördag Datum: 2018-03-10 Tid: 8.30-13.30 (OBS 5 tim) Rum: SB

Ansvarig lärare:	Erland Holmström tel. 1007.
Besökande lärare:	Erland Holmström tel. 1007,
Resultat:	Skickas med mail från Ladok.
Lösningar:	Läggs eventuellt på hemsidan.
Tentagranskning:	Tentan finns efter att resultatet publicerats på vår expedition. Tid för klagomål på rättningen annonseras på hemsidan efter att resultatet är publicerat (eller maila mig så försöker vi hitta en tid).
Betygsgränser:	CTH: 3=28p, 4=38p, 5= 48p, max 60p För betyg 3 skall man ha 10 poäng på del A och10p på del B. Dock kan gränsen variera något beroende på tentans upplägg.
Hjälpmedel:	-

Observera:

- Börja med att läsa igenom alla problem så du kan ställa frågor när jag kommer.
  Jag brukar komma efter ca 1-2 timmar (men det är många salar så det kan bli senare).
- ***Alla svar måste motiveras*** där så är möjligt/lämpligt.
- Skriv läsligt! Rita figurer. Lösningar som är svårlästa bedöms inte.
- Skriv kortfattat och precist.
- Råd och anvisningar som getts under kursen skall följas.
- Program skall skivas i Java, skall vara indenterade och lämpligt kommenterade.
- Börja nya problem på ny sida, dock ej korta delproblem.
- *Lämna inte in tesen med tentan utan behåll den. Lämna inte heller in kladdpapper/skisser.*

Lycka till!



Tipstack till Simon Sundstrom IT2

Del A Teori *Du måste motivera om det går även om det inte står explicit.*

Problem 1. *Uppvärmning:*

Vilken eller vilka av följande påstående är sann(a) och vilken eller vilka är falsk(a) eller svara på frågan. Alla svar skall motiveras.

a) Ett träd är ett specialfall av en DAG som är ett specialfall av en graf. Speciellt så är en DAG en sammankopplad graf utan cykler.

b) Operationen add(...) i klassen ArrayList tar alltid O(1).

(2p)

Problem 2. *Testar: quicksort, komplexitet, lösa rekursionsekvationer*

a) Beskriv sorteringsmetoden quicksort (qs), steg 1+2.
Tänk inte för mycket på detaljer med index.

d) Beskriv qs komplexitet. Du sätter upp och motiverar rekursionsekvationen för best case (bc). Du löser rek.ekvationen för bc (dvs visar hur man gör, alla detaljer behöver inte vara med). Du behöver inte härleda komplexiteten för findpivot och partition utan kan anta att den är O(n) för bägge.

d) I artikeln ”Engineering a Sort Function” (som också implementerats i Javas quicksort) beskrivs 3 förbättringar av quicksort. Beskriv och motivera dom.

(15p)

Problem 3. Testar: Träd, hashtabeller

När man lagrar heltal och vill söka efter dom effektivt så kan man använda hashtabeller eller AVL-träd. Du skall sätta in talen
12, 44, 13, 88, 23, 94, 11, 39, 20, 16 i strukturena nedan.

a. Sätt in talen i given ordning i ett AVL-träd. Du ritar upp trädet alldeles före en balansering och visar hur balanseringen går till och resultatet. Om ingen balansering behövs behöver du inte rita ut trädet för det steget. Du bör också motivera varför du gör just de balanseringar du gör.

b. Sätt in samma tal i en hashtabell som har 11 celler och använder chaining dvs man skapar en länkad lista med kollisionerna. Hashfunktionen är
$\mathrm{h}(\mathrm{i}) = (2\mathrm{i}+5) \bmod 11.$

c. Sätt in samma tal i en hashtabell som har 11 celler och använder closed hashing dvs man lagrar allt i det befintliga fältet. Samma hashf. som ovan.
Använd $\mathrm{nästa\ lediga\ plats} \bmod 11$ som omhashningsfunktion.
Motivera varför elementen ligger där dom ligger.

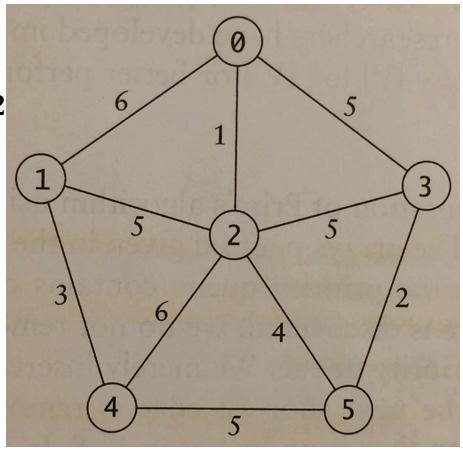
(14p)

Del B Implementeringar och algoritmer

Problem 4. *Testar Grafer, Kruskals algoritm*

a) Beskriv Kruskals algoritm, steg 1 och steg 2 (originalet, inte labversionen)

b) Beskriv hur algoritmen hittar det minimala uppspännande trädet för grafen till höger.



(8p)

Problem 5. *Testar: rekursion över träd.*

a) Definiera vad partiellt ordnat och vänsterbalanserat innebär.

b) Skriv en metod som avgör om ett givet binärt träd är ett partiellt ordnat vänsterbalanserat träd. Det finns ett klass-skal för ett binärt träd nedan. Metoden får inte vara onödigt ineffektiv eller minneskrävande. Du behöver inte bry dig om typparametrar, antag att data (E) är ett heltal. Signatur för en intern metod blir (du behöver inte skriva wrappern och syntaxen behöver inte vara perfekt men andemeningen måste framgå)

```
private boolean isPOV(BinaryTree t) {…}
```

c) Vad är komplexiteten för din metod?

(11p)

Problem 6. *Testar algoritmer, efterföljarlistor*

Antag att vi har en mängd element som tilldelas prioritet i form av ett heltal i ett intervall 1..k, där k är ett litet tal. 1 anger högsta prioritet och k lägsta. Vi kan då implementera en prioritetsköklass som internt har *k* köer, en för varje prioritet. Nya element sätts in sist i den kö som deras prioritetstal anger. (Jämför med en efterföljarlista för grafer).

Om vi använder en LinkedList för de interna köerna (se API i slutet) så hoppas vi att detta skall ge oss operationer med komplexitet O(1) för prioritetsköklassen och där dessutom prioritetskön är stabil som ju inte en POV implementerad kö är. Med stabil menas att bland element med lika prioritet tas det element som väntat längst ut först.

Skissa på lösningen dvs algoritm steg 1. Figurer är bra.

Ge pseudocode för insert och delete (och deklarationer så man förstår och felhantering skall vara med)

Kan man lyckas komma ner till O(1) för både delete och insert?

10p

```
public class BinaryTree<E> {
  protected static class Node <E> {...
    ...data, left, rigth }

/** The root of the binary tree */
  protected Node<E> root;

  public BinaryTree() {...
  protected BinaryTree(Node<E> root) {...
  public BinaryTree(E data,
                    BinaryTree<E> leftTree,
                    BinaryTree<E> rightTree){...

  //Return the left/right subtree.
  public BinaryTree<E> getLeftSubtree(){...
  public BinaryTree<E> getRightSubtree(){...
  //Return the data field of the root
  public E getData() {...

  //Is this tree a leaf.
  public boolean isLeaf() {…
  // Is the tree empty
  public boolean isEmpty() {...
  public String toString() {...
}
```