TDA416 Datastrukturer 2018.03.10 Lösningsskisser

Problem 1. Uppvärmning:

Vilken eller vilka av följande påstående är sann(a) och vilken eller vilka är falsk(a) eller svara på frågan. Försök motivera svaren så gott det går.

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.
(ja) (rita figurer, DAG= träd men tillåter mer än en förälder)

b) Operationen add(...) i klassen ArrayList tar alltid O(1).
(nej, när tabellen blir full så omskapas den och det tar tid)

Problem 2.

a) Beskriv sorteringsmetoden quicksort (qs), steg 1+2. … Se F12 slide 11

d) qs komplexitet. … Se F12 slide 11 samt lösningen på övningen efter F3.

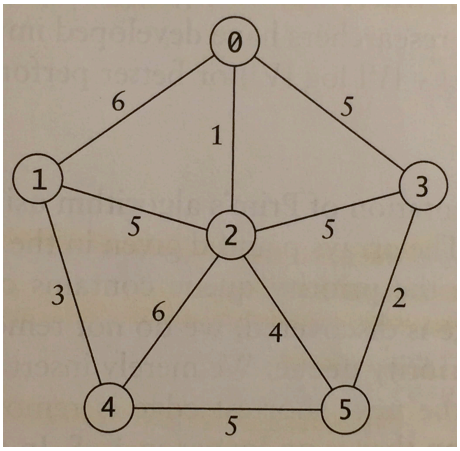
d) I artikeln ”Engineering a Sort Function”... Beskriv och motivera dom.

- små fält (<7) sorteras med insertion sort,
  => inline kod dvs inga metodanrop => mindre overhead
- välj pivotelementet på olika sätt utifrån storlek på fältet
  större fält => noggrannare val
- placera lika element i mitten vid partition
  då undviker vi sned delning vid många lika

Problem 3. *se nästa sida*

Del B Implementeringar och algoritmer

Problem 4. *Testar Grafer, Kruskals*



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

se F11 både slide 6 (med lite kommentarer om Mfsets) och 7 går bra.

b) Beskriv....

Bågarna väljs i kostnadsordning om dom inte skapar en cykel

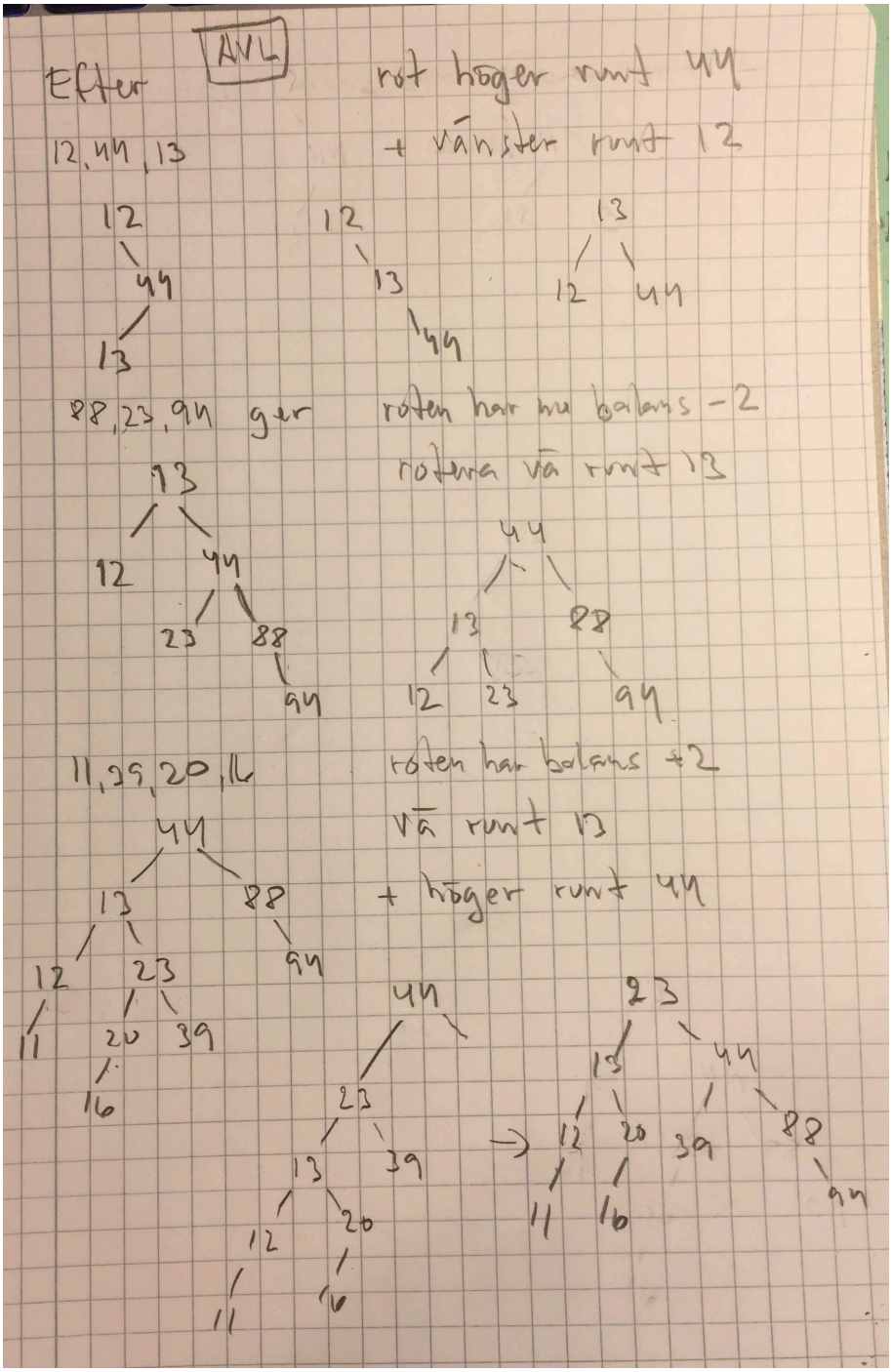
Båge	kostnad	välj
0-2	1	ja
3-5	2	ja
1-4	3	ja
2-5	4	ja
0-3, 2-3,	5	nej cykel
1-2, 4-5	5	ja men bara en av dom

klart

Problem 3:

När man lagrar heltal ... Du skall sätta in talen
12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5 i strukturena nedan.

a) Sätt in talen i given ordning i ett AVL-träd.



b) Sätt in samma tal i en hashtabell som har 11 celler ...$\mathrm{h}(\mathrm{i}) = (2\mathrm{i}+5) \bmod 11$.
Vi får hashkoderna (listorna fylls sedan nerifrån och upp)

vänd...

```
i:      12 44 13 88 23 94 11 39 20 16
h(i):    7  5  9  5  7  6  5  6  1  4
                          11
                          88  39  23
          20           16  44  94  12       13
lista 0   1   2   3   4   5   6   7   8   9   10
```

c) Sätt in samma tal... som har 11 celler och använder closed hashing …

12 läggs direkt i cell 7, 44 läggs direkt i cell 5, 13 läggs direkt i cell 9
88 cell 5 är upptagen, nästa lediga är 6
23 cell 7 är upptagen, nästa lediga är 8
94 cell 6 är upptagen, nästa lediga är 10
11 cell 5 är upptagen, nästa lediga är 0
39 cell 6 är upptagen, nästa lediga är 1
20 cell 1 är upptagen, nästa lediga är 2
16 läggs direkt i cell 4

```
11 39 20 -- 16 44 88 12 23 13 94
 0  1  2  3  4  5  6  7  8  9 10
```

Problem 5. *Testar: rekursion över träd.*

a) Definiera vad pov.... Se F6+7 slide 5+6

b) Skriv en metod som ... är ett partiellt ordnat vänsterbalanserat träd...

Det är som vanligt ett elände att få alla Java-detaljer rätt. Pseuodokod räcker dock för er. Vi behöver kolla två saker: att trädet är **partiellt ordnat och att det är vänsterbalanserat**. Man kan kanske göra bägge på samma gång men jag föredrar att dela upp det i två metoder eftersom den ena görs enklast med dfs och den andra med bfs.

```
boolean isPOV(BinaryTree t) {
  return isLeftBalance(t) && isPO(t);
}
boolean isPO(BinaryTree<String> t) {
  if( t==null && t.isEmpty() )
    return true;
  BinaryTree<String> tleft  =  t.getLeftSubtree();
  BinaryTree<String> tright =  t.getRightSubtree();
  boolean leftOk = true;
  boolean rightOk = true;
  if( tleft!=null  ) {
    leftOk = Integer.valueOf(t.root.data) <=
              Integer.valueOf(tleft.root.data)
              && isPO(tleft);
  }
  if( tright!=null ) {
    rightOk = Integer.valueOf(t.root.data) <=
               Integer.valueOf(tright.root.data)
              && isPO(tright);
  }
  return leftOk && rightOk;
} // end isPO
```

isLeftBalance gör en bfs genomlöpning av trädet.

Normalläget skall vara 2 barn. När vi stöter på en nod med 0 barn eller 1 vänsterbarn så skall alla noder sedan ha 0 barn. Alla andra lägen är fel.

```
boolean isLeftBalance(BinaryTree<String> t) {
  boolean stillTwoChildren = true;
  Queue<BinaryTree<String>> q =
                        new LinkedList<BinaryTree<String>>();
  q.offer(t);
  while ( !q.isEmpty() ) {
    BinaryTree<String> tmp = q.poll();
    if (tmp != null){
      BinaryTree<String> left  = tmp.getLeftSubtree();
      BinaryTree<String> right = tmp.getRightSubtree();
      if ( !stillTwoChildren &&  // we must have 0 children
           (left != null || right != null) ) {
        return false;
      // here we can have two children or one left child
      } else if ( (left != null && right != null) ) { // two
        q.offer(left);
        q.offer(right);
      } else if ( left != null ) { // one left child also ok
          // but from now on we need zero children
          stillTwoChildren = false;
        q.offer(left);
      } else if ( right != null ) { // one right child not ok
        return false;
      } else { // else new turn
        ;
      }
    }
  }
  return true;
} // end dfs
```

c) Vad är komplexiteten för din metod?

IsPO: $T(n) = 2T(n/2) + c \in O(n)$

Kön i isLeftBalance implementeras med en LinkedList som gör både insättning och uttag på konstant tid. Vi besöker alla noder i trädet genom att lägga in dom i kön.

$T(n) = n$

Tillsammans $n+n \in O(n)$

Problem 6. Testar algoritmer

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.....

Ja vi kommer ner till O(k) där k är antalet prioriteter. Det är metoden firstNonEmptyIndex som anropas i delete som styr. Den måste söka igenom fältet efter en ickse tom lista och det tar max O(k) eftersom fältet är k långt.

Insert ∈ O(1), delete ∈ O(k)

Eftersom elementen ska placeras in i rätt kö efter prioritet väljer jag att de objekt som ska sättas in i kön implementerar interfacet Priority: (men det kan göras på många fler sätt, tex två parametrar)

```
public interface Priority {
  /**
  * Returns the element's priority
  * @return the element's priority
  */
  int priority();
  /**
  * Sets the element's priority to prio
  * @param prio the new priority
  */
  void setPriority(int prio);
}
```

```
import java.util.LinkedList;
public class SimplePriorityQueue<E extends Priority> {
  private LinkedList<E>[] queue;
  private int size;
  /**
  * Creates an empty priority queue.
  * The allowed priorities are in the interval [1..k]
  * @param k the highest allowed priority
  */ ======================================
  public SimplePriorityQueue(int k) {
    queue = (LinkedList<E>[]) new LinkedList[k + 1];
    for(int i=1; i<queue.length; i++) {
      queue[i] = new LinkedList<E>();
    }
    queue[0] = null; // plats 0 används ej
    size = 0;
  }

  /** ======================================
  * Inserts the specified element into this priority queue.
  * @param e the element to add
  * @return true
  * @throws NullPointerException if the specified elem is null
  * @throws IllegalArgumentException if the elements prio is
  * outside the allowed interval
*/
```

```
  public boolean insert(E e) {
    if (e == null) {
      throw new NullPointerException();
    }
    int prio = e.priority();
    if (prio <= 0 || prio >= queue.length) {
      throw new IllegalArgumentException();
    }
    queue[prio].add(e);
    size++;
    return true;
  }

  /** ========================================
  * Retrieves and removes the smallest element in this queue,
  * or returns null if this queue is empty.
  * @return the smallest element in this queue, or null if
  * this queue is empty
  */
  public E delete() {
    // this is the only complexity problem
    int index = firstNonEmptyIndex();
    if (index == -1) {
      return null;
    }
    size--;
    return queue[index].remove();
  }

  /*========================================
  * Returns the index for the first non empty list,
  * or returns -1 if all lists are empty.
  * complexity is O(k)
  */
  private int firstNonEmptyIndex() {
    for (int i = 1; i < queue.length; i++) {
      if (queue[i].size() != 0) {
        return i;
      }
    }
    return -1;
  }

  /** ========================================
  * Returns the number of elements in this queue.
  */
  public int size() {
    return size;
  }
} // end SimplePriorityQueue
```

```
public class Main {      //behövdes ej på tentan

  public static void main(String[] args) {
    class PatternForE implements Priority {
      int prio = 0;
      String data = "";
      public PatternForE(int prio, String data) {
        this.prio = prio;
        this.data = data;
      }

      public int priority() {
        return prio;
      }
      public void setPriority(int prio) {
        this.prio = prio;
      }
      public String toString () {
        return "* " + prio + " " + data + " *";
      }
    }

    PatternForE p1 = new PatternForE(1, "ett");
    PatternForE p21 = new PatternForE(2, "två");
    PatternForE p22 = new PatternForE(2, "två*2");
    PatternForE p23 = new PatternForE(2, "två*3");
    PatternForE p31 = new PatternForE(3, "tre*1");
    PatternForE p32 = new PatternForE(3, "tre*2");
    PatternForE p33 = new PatternForE(3, "tre*3");
    PatternForE p34 = new PatternForE(3, "tre*4");
    PatternForE p4 = new PatternForE(4, "fyra");
    PatternForE p5 = new PatternForE(5, "fem");
    PatternForE p6 = new PatternForE(6, "sex");

    SimplePriorityQueue<PatternForE> pq =
                                new SimplePriorityQueue(5);
    pq.insert(p23); pq.insert(p31);
    pq.insert(p22); pq.insert(p21);
    pq.insert(p32); pq.insert(p33);
    pq.insert(p5); pq.insert(p34);
    System.out.println(pq.delete());
    System.out.println(pq.delete());
    System.out.println(pq.delete());
    ….
    System.out.println(pq.delete());

  }
}
```