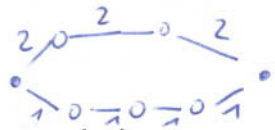
Kortfattade lösningar till tentan i TDA416 2013-03-12 – Birgit Grohe

Problem 1:

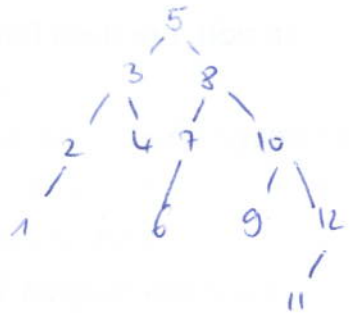
a) Nej. Binära sökträd kan vara obalanserade.
b) Nej. Att nå mitten-elementet i en länkat lista tar redan O(n), därför är binärsökning inget effektivt sätt att hitta elementet.
c) Stack: 20.
d) Kö: 15 och 60 (15 längst fram)
e) Nej. Primtal ger bättre spridning/färre kollisioner.



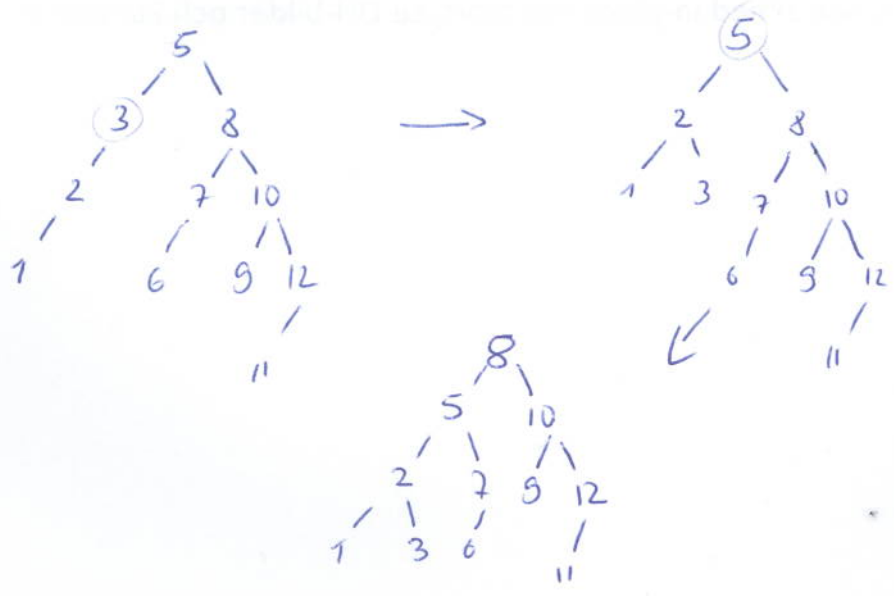
f) Nej. Motexempel: Två vägar mellan en startnod och en slutnod; den ena vägen består av 3 bågar med kostnad 2 och den andra av 4 bågar av kostnad 1.
g) Om det finns ett unikt MST, så är svaret ja. I annat fall så Kruskals och Prims returnera olika MST (dock med samma totalvärde), beroende på implementationen.
h) Ja. Båda har O(n^2), men quicksort har O(n log n) i average case.

Problem 2:

a).



b).

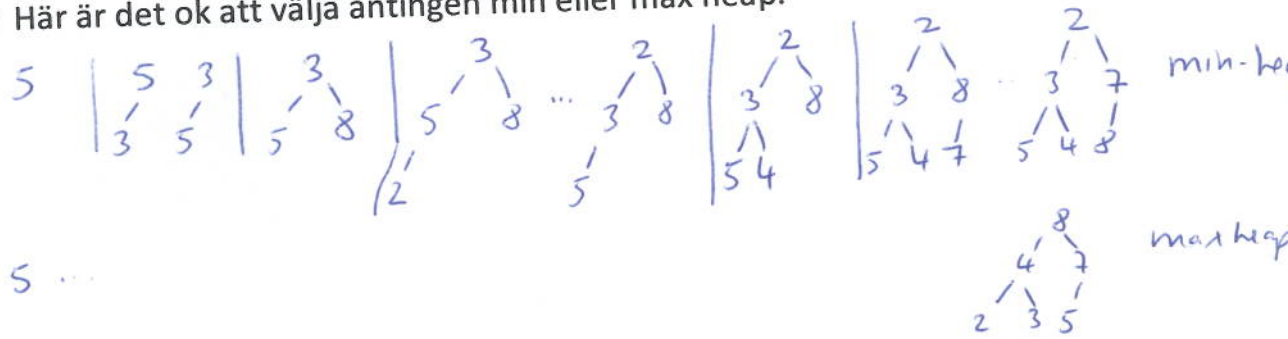


c) AVL-träd är höjdbalanserade så att insättning, borttagning och sökning ger O(log n). Sökning är samma för båda träd, men för remove och insert så finns det rotationer inbyggda för AVL-träd så att trädet stannar höjdbalanserad. För detaljer se kursboken och OH-bilderna.

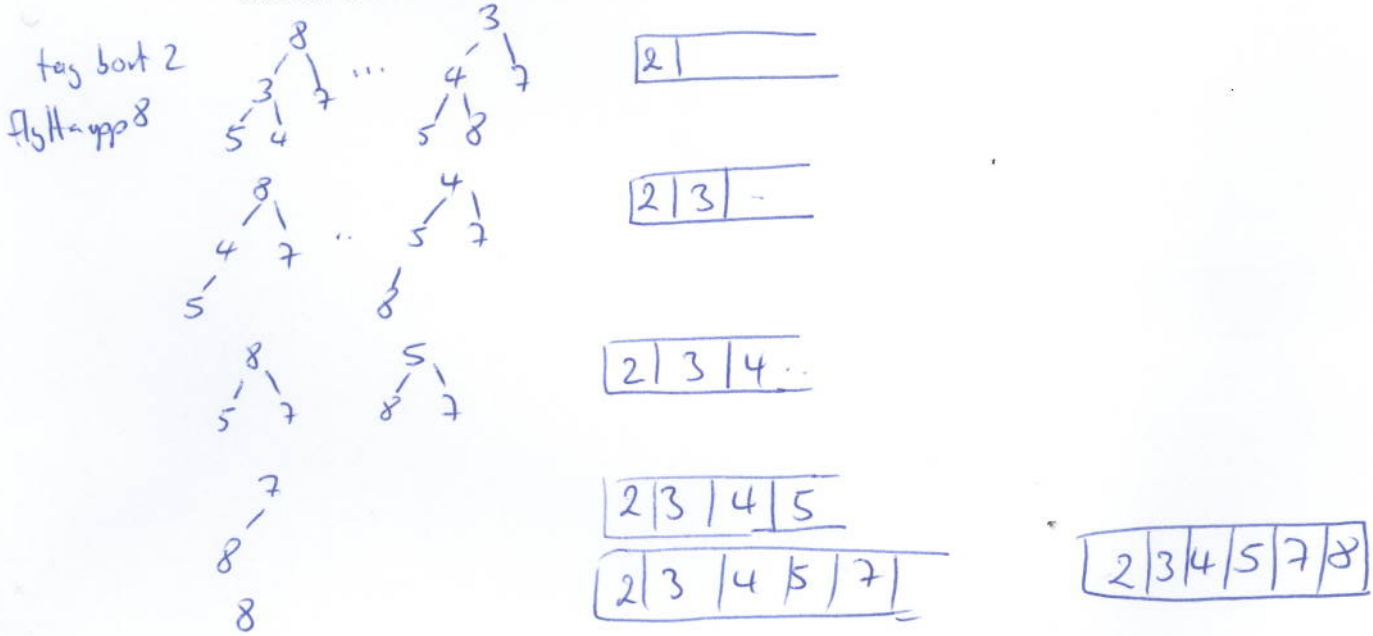
Problem 3:

a) POV är ett träd där barnen är större än föräldrarna (min heap). Det finns också max-heap där barnen är mindre än föräldrarna. Dessutom är det ett binärt träd (max två barn) och vänsterbalaserad betyder att det alla nivåer är fulla förutom sista nivån, som är fylld från vänster.

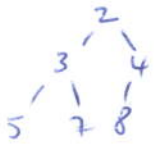
b) Här är det ok att välja antingen min eller max heap.



c) Roten ligger på position p=0 i fältet. Barnen till en nod, om dom finns, ligger på position 2p+1 och 2p+2.
Heapsort med O(n) extra minne: Börja med en heap och ett tomt fält. Tag ut minsta elementet från heapen (med deleteMin) och lägg i fältet på första ännu ej upptagna plats. Flytt upp sista elementet av den kvarvarande heapen till roten. Återställ heap-egenskapen. Fortsätt tills heapen är tom och fältet fullt.
Det är ok att svara med in-place heapsort, se OH-bilder och kursboken.



d) Det sorterade fältet i c) är [2,3,4,5,7,8] vilket ger följande träd:

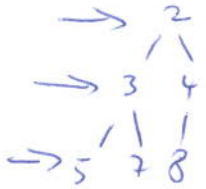


Preorder treversal av detta träd ger: 2, 3,5,7, 4, 8

e) Börja med roten och tag sedan noderna i nästa nivå från vänster till höger. Fortsätt att gå genom trädet nivå efter nivå. Nedan kortfattat pseudokod:

```
kö k; k.offer(root);
while( kö not empty ){
        node n= kö.pop();
        print n;
        if( exists n.leftChild ) kö.offer(n.leftChild);
        if( exists n.rightChild) kö.offer(n.rightChild);

}
```



Problem 4: Tal/hashcode

40	16	90	38	7	72	82	28	12	59
7	5	2	5	7	6	5	6	1	4

a) chaining

0	
1	12
2	90
3	
4	59
5	16,38,82
6	72,28
7	40,7
8	
9	
10	

b) Open adressing med linear probing

0	28
1	12
2	
3	
4	59
5	16
6	38
7	40
8	7
9	72
10	82

c) Först utför man en sökning efter elementet. Gå till tabellens position som motsvarar hashkoden. Om talet finns där, så är man klar. Om det finns ett annat tal där eller en markör för dummy/gravsten (dvs det har funnits ett tal där tidigare som tagits bort), så fortsätt ett steg till höger (evtl wrap-around). Gör detta så länge tills du antingen hittar talet, eller tills du komma till en tom cell. Om du hittar talet, tag bort talet och sätt in dummy markören. Annars finns inte talet i tabellen och då kan man inte göra remove.

d) Beskrivning se c). Borttagning av 82 och 28 (inte 18 som det står i tesen).

Problem 5:

a) 3
b) Gör en BFS för varje nod v i nod-setet V. Då får man svar på det minsta avståndet från varje enskild nod har till alla andra noder, låt detta vara diam(v). Tag nu maximum över alla noder max_{v in V} diam(v).
Ett mindre effektivt sätt är att köra Dijkstras algorithm för varje nod (vikt = 1 för varje båge).
c) BFS har O(n+m). Man utför BFS n gånger. O(n^2 +nm) totalt. Man kan också skriva O(mn), då grafen är sammanhängande och antal noder n är O(m).

```java

//Uppgift 6
//==========
public class Poly {
    private LinkedNode first = new LinkedNode(); // dummy first node
    private LinkedNode last = first;

    public Poly() {
        ; // empty
    }

    public void add(int koeff, int expo) {
        last.next = new LinkedNode<Term>();
        last = last.next;
        last.term = new Term(koeff, expo);
    }

    private Poly(LinkedNode n) { // konstruktor som behövs i addPoly
        first = n;
        last = first;
        while (last.next != null) {
            last = last.next;
        }
    }

    public Poly addPoly(Poly rhs) {
        if (rhs == null) {
            throw new NullPointerException(); // eller returnera this
        }
        LinkedNode<Term> newPoly = new LinkedNode<Term>(); // dummy first node
        LinkedNode<Term> itNew = newPoly;
        LinkedNode<Term> it1 = first.next; // avoid dummy node
        LinkedNode<Term> it2 = rhs.first.next;

        while ( it1 != null && it2 != null  ) {
            itNew.next = new LinkedNode<Term>();
            itNew = itNew.next;
            int comparison = it1.term.compareTo(it2.term);
            if ( comparison < 0  ) { // it2 is larger
                Term newT = new Term(it2.term.koeff, it2.term.expo);
                itNew.term = newT;
                it2 = it2.next;
            } else if ( comparison > 0  ) { // it1 is larger
                Term newT = new Term(it1.term.koeff,  it1.term.expo);
                itNew.term = newT;
                it1 = it1.next;
            } else { // equal
                Term newT = new Term(it1.term.koeff + it2.term.koeff, it1.term.expo);
                itNew.term = newT;
                it1 = it1.next;
                it2 = it2.next;
            }
        }
        // minst en av it1 och it2 är nu null
        LinkedNode<Term> remaining = (it1 == null ? it2 : it1);
        while (remaining != null) {
            itNew.next = new LinkedNode<Term>();
            itNew = itNew.next;
            Term newT = new Term(remaining.term.koeff, remaining.term.expo);
            itNew.term = newT;
            remaining = remaining.next;
        }
        return new Poly(newPoly);
    }


    public String toString() {
        LinkedNode<Term> it = first.next;
        if (it == null) {
```

```
70              return "[ ]";
71          }
72          StringBuilder sb = new StringBuilder();
73          sb.append("[");
74          while (it != null) {
75              sb.append(it.term.koeff);
76              sb.append("x");
77              sb.append(it.term.expo);
78              sb.append("+");
79              it = it.next;
80          }
81          int last = sb.lastIndexOf("+");
82          sb.setCharAt(last, ']'); // overwrite the last "+" sign
83          return sb.toString();
84      }
85
86      private class LinkedNode<Term> {
87          public Term term;
88          public LinkedNode<Term> next;
89      }
90
91      static private class Term implements Comparable<Term> {
92          public int expo = 0;
93          public int koeff = 0;
94          public Term(int koeff, int expo) {
95              this.expo = expo;
96              this.koeff = koeff;
97          }
98
99          public int compareTo( Term rhs ) {
100             if ( this == rhs ) return 0;
101             if (this.expo < rhs.expo) return -1;
102             if (this.expo > rhs.expo) return +1;
103             return 0;
104         }
105     } // end Term
106
107     public static void main(String[] args) {
108         Poly p1 = new Poly();
109         p1.add(3,4); p1.add(2,2); p1.add(3,1); p1.add(7,0);
110         Poly p2 = new Poly();
111         p2.add(2,3); p2.add(4,1); p2.add(5,0);
112         System.out.println(p1);
113         System.out.println(p2);
114         System.out.println();
115         Poly p3 = p1.addPoly(p2);
116         System.out.println(p3);
117     }
118 } // end Poly
119
120
```