Institutionen för
Datavetenskap
Göteborgs universitet

VT10
DIT960
24/5 -10

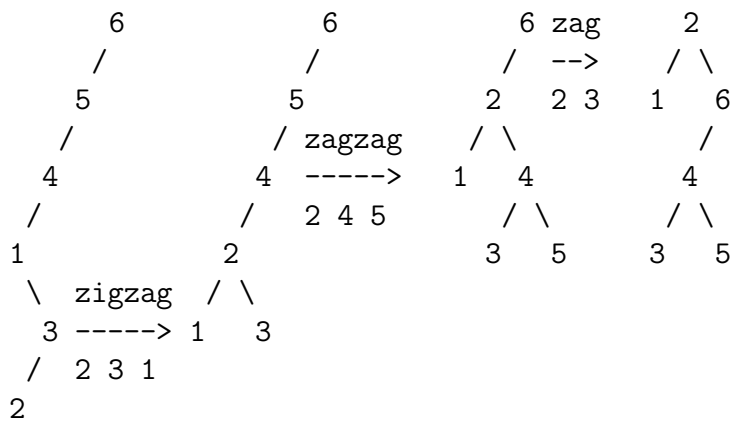
Lösningsförslag till tentamen för Datastrukturer DV

DAG : 31 maj 2010

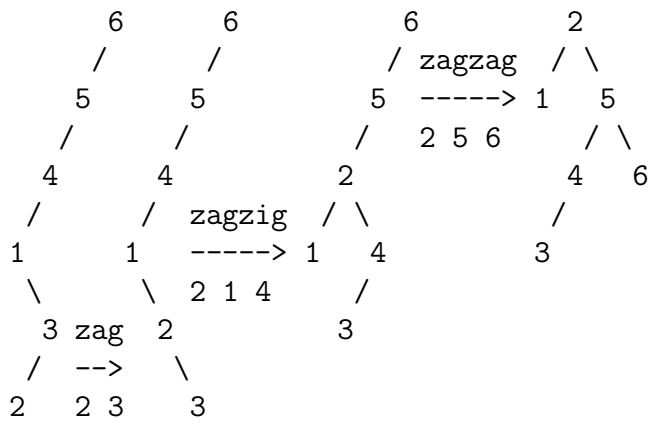
Del A

Datastrukturer på abstrakt nivå.

Uppg 1: a)



alternativt



b)

```
    6         5             5            3
   <  RR     / \           < \   RR    /   \
  5   -->   4   6         3   6  --> 1       5
 /    6 5  <      DRR    / \     5 3  \     / \
4         1       ----> 1   4          2   4   6
           \      4 1 3  \
            3             2
```

c)

```
      s6         s5             s5
     /          /  \           /  \
   r5  RR     r4     r6 flip s4     s6
  /    -->                   /         DRR
r4     6 5                 r1         ---->
                             \        4 1 3
                              r3
```

```
        s5               s5
       /  \             /  \
     s3    s6  flip  r3    s6
     / \              / \
   r1   r4           s1   s4
                      \
                       r2
```

Uppg 2: **a)** Falskt, eftersom ett splayträd kan vara hur skevt som helst.

b) Falskt, höjden på delträden i ett AVL-träd får ju skilja högst 1, men i ett rödsvart träd kan ju skillnaden vara betydligt större.

c) Falskt. Det tar $\mathbf{0}(n)$ att dela listan på hälften, dvs om listan är sorterad:

$$\sum_{i=1}^{^2log\ n} \frac{n}{2^i} \quad \in \quad \mathbf{O}(n)$$

d) Sant, att hitta kortaste väg enligt Dijkstra's metod innebär ju i princip att hitta ALLA kortaste vägar och sammanhängande innebär ju att du från en nod har minst en väg till varje annan nod.

e) Sant. Första element är ju sista elementets 'nästa element'.

f) Falskt. T.ex. om fältet redan är sorterat är arbetet av $\mathbf{O}(\,n\,)$, medan om fältet är omvänt sorterat är komplexiteten av $\mathbf{O}(n^2\,)$

Uppg 3: På raden under elementen anges med ett p vilka element som är pivot-element, och index anger med vilket index som swap har gjorts.

Maximala rekursionsdjupet blir 6, där den sista ej ritas ut eftersom inget händer,

0	1	2	3	4	5	6	7	8	9	10	11
80	90	110	40	30	60	100	50	120	10	20	70
10	9	7				11	2		1	0	p6
20	10	50	40	30	60	70	110	120	90	80	100
					p5	-	10	9	8,11	7	p9
20	10	50	40	30	60	70	80	90	100	110	120
		4		p2	-	-		p8	-		p11
20	10	30	40	50	60	70	80	90	100	110	120
1	p0	-		p4	-	-		-	-		-
10	20	30	40	50	60	70	80	90	100	110	120
-		-		-	-	-	-	-	-	-	-

Notera att det är onödigt att flytta pivot-elementet längst till vänster, eftersom det redan 'är ur vägen'. Däremot har jag inte dragit av poäng för de som flyttade pivo-telementet längst till vänster i delsegmentet, även om det är onödigt.

Uppg 4: **a)** Låt *nod* vara odefinierad.
Låt *min* vara oändligheten.
Låt *nl* vara den givna nodlistan.

För varje n i *nl*
    låt *sum* vara 0
    för varje k i *nl*
        låt *sp* vara kortaste vägen mellan n och k
        lägg till alla vikterna i *sp* till *sum*
    Om $sum < min$
        låt *nod* och *min* vara n respektive *sum*

Returnera *nod*

b) Om komplexiteten för kortaste vägen är $\mathbf{O}(\ |E|\, log\, |E|\ )$ blir komplexiteten $\mathbf{O}(\ n^2 * |E|\, log\, |E|\ )$, dvs värstafallskomplexiteten blir $\mathbf{O}(\ |V|^2 * |E|\, log\, |E|\ )$. n, $|V|$ och $|E|$ är antalet noder i den givna listan, antalet noder i grafen respektive antalet bågar i grafen.

Uppg 5:

```
public int centerNode( List<Integer> li ) {

   int node = -1;
   int min  = Integer.MAX_VALUE;

   for ( int from : li ) {
      int sum = 0;
      for ( int to : li ) {
         Iterator<E> it = shortestPath( from, to );
         while ( it.hasNext() )
            sum += it.next().weight();
         }
         if ( sum < min ) {
            min  = sum;
            node = from;
         }
      }
   return node;

} // centerNode
```

Uppg 6: **a)**

```
import BinSearchTree

contains :: Ord a => a -> BST a -> Bool
contains a at
 | isEmpty at = False
 | a < root  = contains a (leftSub at)
 | a > root  = contains a (rightSub at)
 | otherwise = True
 where root = rootVal at
```

Som har värstafallskomplexiteten $\mathbf{0}(h)$, där h är trädets höjd.

Samma komplexitet har, om `get` är vettigt definierad:

```
import BinSearchTree

contains :: Ord a => a -> BST a -> Bool
contains a at = get a at == Just a
```

(Det var egentligen inte meningen att get skulle vara med, men jag glömde stryka den funktionen.)
Alternativt kan vi använda den ineffektivare metoden, som har värstafallskomplexiteten $\mathbf{0}(n)$, där n är antalet noder i trädet :

```
import BinSearchTree

contains :: Eq a => a -> BST a -> Bool
contains a at = or [  a == x | x <- inorder at ]
```

b)

```
import BinSearchTree

postOrder :: BST a -> [a]
postOrder at
 | isEmpty at = []
 | otherwise  = posto at []

posto :: BST a -> [a] -> [a]
posto at as
 | isEmpty at = as
 | otherwise  = posto (leftSub at) $
                      posto (rightSub at) $
                            rootVal at : as
```

Givetvis kan vi använda en stack för att 'lagra svansen':

```
import BinSearchTree
import Stack

postOrder :: BST a -> [a]
postOrder at
 | BinSearchTree.isEmpty at = []
 | otherwise  = toList $ post at emptyStack

post :: BST a -> Stack a -> Stack a
post at as
  | BinSearchTree.isEmpty at = as
  | otherwise = post (leftSub at) $
                     post (rightSub at)
                          (push (rootVal at) as)

toList :: Stack a -> [a]
toList as
  | Stack.isEmpty as = []
  | otherwise        = top as : pop as
```

Uppg 7:

```
private class BinTreeIterator
        implements Iterator<E> {

  Stacks<TreeNode<E>> nextOnTop;
  TreeNode<E>         lastNext;
  boolean             remAllowed;

  public BinTreeIterator() {
    nextOnTop  = new LinkedStack<TreeNode<E>>();
    lastNext   = null;
    remAllowed = false;
    if ( root != null )
      nextOnTop.push( root );
    }

    public boolean hasNext() {
      return ! nextOnTop.isEmpty();
    } // hasNext

    public E next() {
      lastNext = nextOnTop.pop();
      if ( lastNext.right != null )
        nextOnTop.push(lastNext.right);
      if ( lastNext.left != null )
        nextOnTop.push(lastNext.left);
        remAllowed = true;
        return lastNext.element;
    } // remove

    public void remove() {
      if (remAllowed) {
        remAllowed = false;
        remove( lastNext ); // se nästa sida
      }
      else
         throw new IllegalStateException();
    }  // remove public
```

```
  private void remove( TreeNode<E> t ) {
    if ( t.left == null )
     // Om inget vänsterträd, länka bara förbi
     // och ändra fader-noden för höger-sonen.
     // Eventuell höger-son ligger redan överst
     // i stacken.

      if ( t.parent == null ) {
        root = t.right;
        if ( t.right != null )
          t.right.parent = null;
      }
      else {
         if (t.parent.left == t )
            t.parent.left = t.right;
         else
           t.parent.right = t.right;
         if ( t.right != null )
           t.right.parent = t.parent;
      }
    else {
      // Kopiera in hela vänsterkedjans element
      // ett steg uppåt och låt understa höger-
      // sonen bli vänsterson. Justera stacken
      // så att givna nod t åter ligger överst
      // på stacken.

      nextOnTop.pop();
      if (t.right != null)
         nextOnTop.pop();
      nextOnTop.push( t );
      t.element = t.left.element;
      while ( t.left.left != null ) {
        t = t.left;
        t.element = t.left.element;
      }
      t.left = t.left.right;
    }
  } // remove private
}
```

Ett litet testprogram, ingick ej i uppgiften.

```
    public static void main( String[] args ) {

     // Konstruera exemplet i tentamenstesen
        BinTreeWithParent<Integer> tree
            = new  BinTreeWithParent<Integer>();
        TreeNode<Integer> node5
         = new TreeNode<Integer>(5,null,null,null);
        TreeNode<Integer> node2
         = new TreeNode<Integer>(2,null,null,node5);
        node5.left = node2;
        node2.left
         = new TreeNode<Integer>(1,null,null,node2);
        TreeNode<Integer>  node4
         =  new TreeNode<Integer>(4,null,null,node2);
        node2.right = node4;
        node4.left
         = new TreeNode<Integer>(3,null,null,node4);
        TreeNode<Integer> node6
         =  new TreeNode<Integer>(6,null,null,node5);
        node5.right = node6;
        node6.right
          = new TreeNode<Integer>(7,null,null,node6);
        tree.root = node5;
     // Skriv ut noderna i preorder.
        Iterator<Integer> it = tree.iterator();
        while ( it.hasNext() ) {
          int i = it.next(); System.out.print(" "+i);
        }
        System.out.println();
     // Gå igenom hela trädet och ta bort varje
     // nod och skriv ut noderna som tas bort.
        Iterator<Integer> it = tree.iterator();
        while ( it.hasNext() ) {
          int i = it.next(); it.remove();
          System.out.print( " " + i );
        }
        System.out.println();
    }
  } // class BinTreeWithParent
```