Institutionen för
Datavetenskap
Göteborgs universitet

VT09
DIT960
20/8 -09

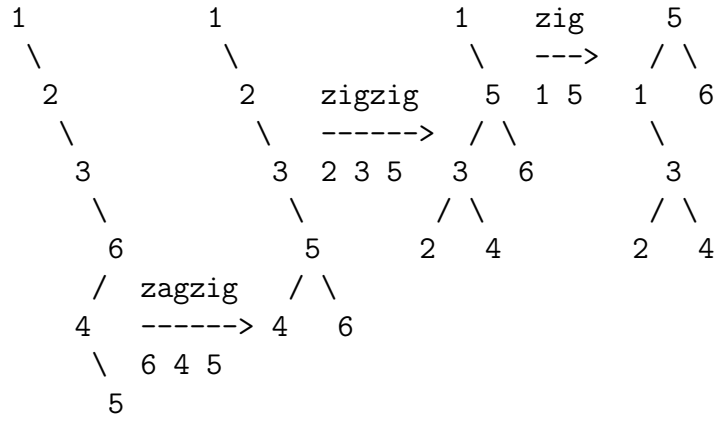
Lösningsförslag till tentamen för Datastrukturer DV

DAG : 20 augusti 2009

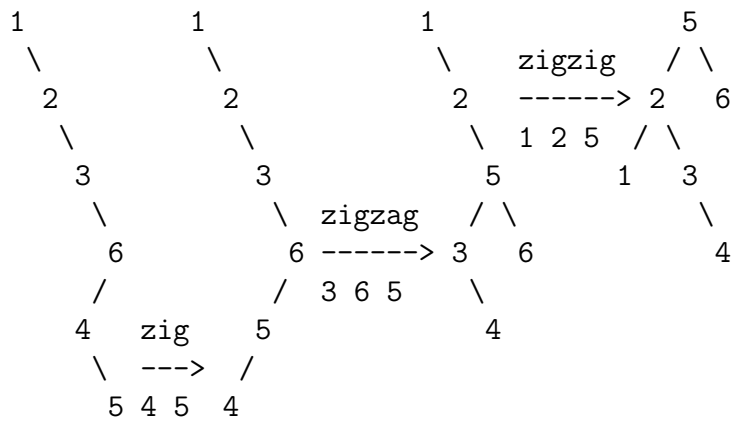
Del A

Datastrukturer på abstrakt nivå.

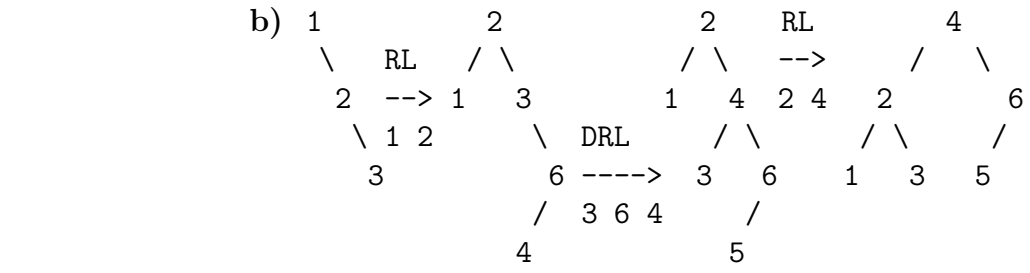
Uppg 1: **a)**



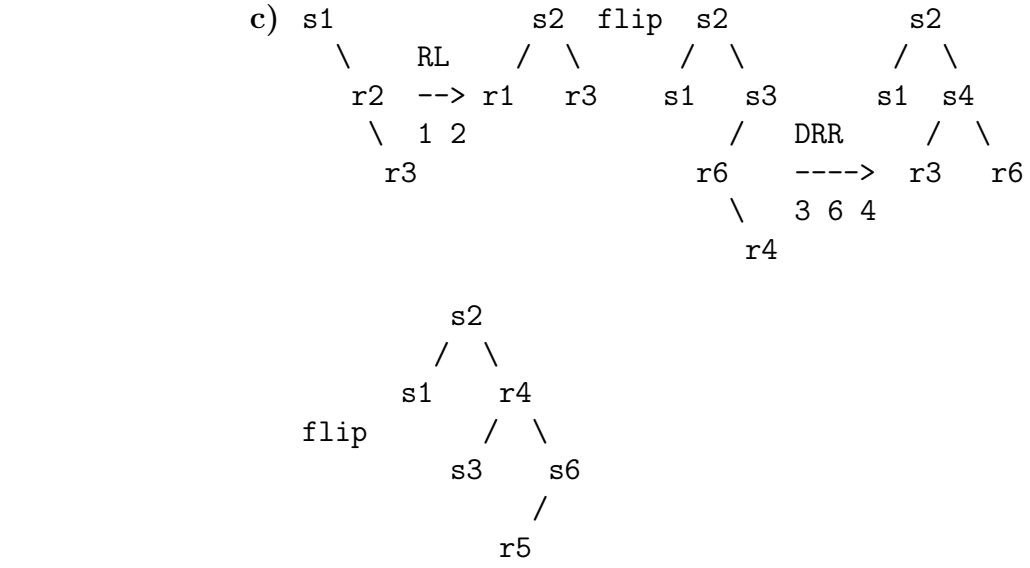
alternativt



b)


1 2 2 RL 4
\ RL / \ / \ --> / \
2 --> 1 3 1 4 2 4 2 6
\ 1 2 \ DRL / \ / \ /
3 6 ----> 3 6 1 3 5
/ 3 6 4 /
4 5

c)


s1 s2 flip s2 s2
\ RL / \ / \ / \
r2 --> r1 r3 s1 s3 s1 s4
\ 1 2 / DRR / \
r3 r6 ----> r3 r6
\ 3 6 4
r4
s2
/ \
s1 r4
flip / \
s3 s6
/
r5

Uppg 2:

a) Falskt. Ett binärt sökträd har en värstafalls-komplexit av $\mathbf{O}(n)$, medan ett AVL-träd har en värstafalls-komplexitet av $\mathbf{O}(^2log\, n)$

b) Falskt. Grafen måste vara sammanhängande.

c) Sant. Antingen har man en länk till sista elementet i listan eller så gör man den cirkulär.

d) Sant. Om man inte vet hur nycklarna ser ut (eller deras spridning), så är lägsta värstafalls-komplexiteten av $\mathbf{O}(n * {}^2log\, n)$, vilket ju gäller för sammansmältningssortering.

e) Sant. Säg, att vi söker efter det första elementet i fältet !

f) Falskt. 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	p 6
20	10	50	40	30	60	70	110	120	90	80	100
					p 5	-	10	9	8,11	7	p 9
20	10	50	40	30	60	70	80	90	100	110	120
		4		p 2	-	-		p 8	-		p 11
20	10	30	40	50	60	70	80	90	100	110	120
1	p 0	-		p 4	-	-		-	-		-
10	20	30	40	50	60	70	80	90	100	110	120
-		-		-	-	-	-	-	-	-	-

Uppg 4: **a)** Detta är en uppgift som är som klippt och skuren för en bredden först traversering i en graf !

Vi vet inget om vilken implementeringen av den graf vi skall hantera, men vi kan anta att varje nod har en färg (svart eller vit).

Låt *que* vara en tom kö.
Låt *col* vara en tom samling.
Lägg startnoden (med satt färg) i *que* och *col*.
Så länge kön ej är tom
    Låt x vara det första element i *que*
    För x:s alla grannar y
        Om y är i *col* och y:s färg är lika med x:s färg
            Returnera att grafen ej kan färgläggas
        Annars, om y ej i *col* låt den få den andra färgen
                och lägg den sedan i *que* och *col*
    Ta bort x ur kön.
Returnera grafen

b) Varje båge hanteras, dvs minst $\mathbf{0}(|E|)$. För varje båge kollas om grannen är i *col*, som är av $\mathbf{0}(\, f(|V|)\,)$ där f beror på samlingen *col*. (Vi kan anta att addera ett element till *col* ej har sämre komplexitet.)

Totalt blir det alltså: $\mathbf{0}(\, |E| \, * \, f(|V|)\,)$

Uppg 5: Lämpligen användes ömsesidig rekursion och en kö, eftersom man måste bryta mot trädets rekursiva struktur.

```
import BinTree
import Queue

breadthFirst :: BT a -> [a]
breadthFirst t = bF t emptyQueue
  where bF t que
         | BinTree.isEmpty t = checkQue que
         | otherwise = rootVal t :
                         checkQue (enqueue (rightSub t)
                                           (enqueue (leftSub t) que))
        checkQue que
            | Queue.isEmpty que = []
            | otherwise         = bF (front que) (dequeue que)
```

Uppg 6: **a)**

```
public boolean add(E elem) {
   if (last == null) {
      last = new Entry( elem, null );
      last.next = last;
   }
   else {
      last.next = new Entry( elem, last.next );
   }
   return true;
} // add
```

b)

```
public int size() {
   if ( last == null )
      return 0;
   else {
      int size = 1;
      Entry p = last.next;
      while ( p != last ) {
         size++;
         p = p.next;
      }
      return size;
   }
} // size
```

c)

```
public boolean removeLast() {
   if (last == null)
      throw new NoSuchElementException();
   else if ( last.next = last )
      last = null;
   else {
      Entry p = last.next;
      while ( p.next != last )
          p = p.next;
      p.next = p.next.next;
      last = p;
   }
} // removeLast
```

d) `add` arbetar enbart med noden som `last` refererar till, dvs $\mathbf{O}(1)$.

`removeLast` måste först leta upp den näst sista noden genom att följa länkarna 1 varv, varför komplexiteten blir av $\mathbf{O}(n)$

`size` följer länkarna 1 varv, dvs $\mathbf{O}(n)$

Uppg 7:

```
public Iterator<Edge>
       shortestPath( int startNode, int endNode ) {

  boolean[] visited     = new boolean[graph.length];
            // har kortaste vägen hittats ?
  int[]     pred        = new int[graph.length];
            // Från vilken nod går den kortaste vägen
  int[]     minW        = new int[graph.length];
            // hur lång är den totala kortaste vägen
  boolean   notFinished = true;

  Arrays.fill( pred, -1 );
  Arrays.fill( minW, Integer.MAX_VALUE );
  minW[startNode] = 0;

  while( notFinished ) {
    //  Hitta noden med lägst vikt bland de icke avklarade
    int mP   = -1;
    int minPathLength = Integer.MAX_VALUE;
    for ( int i = 0; i < graph.length; i++ )
      if( ! visited[i] &&
          minW[i] < minPathLength ) {
        mP = i;
        minPathLength = minW[i];
      }

    // Uppdatera visited, pred och minW för
    // alla grannar till mP

    if ( mP == endNode ||  mP == -1 )
      notFinished = false;
    else {
      visited[mP] = true;
      for( int i = 0; i < graph.length; i++ )
         if ( graph[mP][i] > -1 &&
               minW[mP] + graph[mP][i] < minW[i] ) {
           minW[i] = minW[mP] + graph[mP][i];
           pred[i] = mP;
         }
      }
  }
```

```
  // Gå igenom pred utifrån endNode och skapa en lista.

  List<Edge> path = new LinkedList<Edge>();
  if ( minW[endNode] < Integer.MAX_VALUE ) {
    int n = endNode;
    while ( n != startNode ) {
      path.add( new Edge( pred[n], n, graph[pred[n]][n] ) );
      n = pred[n];
    }
  }
  return path.iterator();

} // shortestPath
```