Institutionen för
Datavetenskap
Göteborgs universitet

VT09
DIT960
25/5 -09

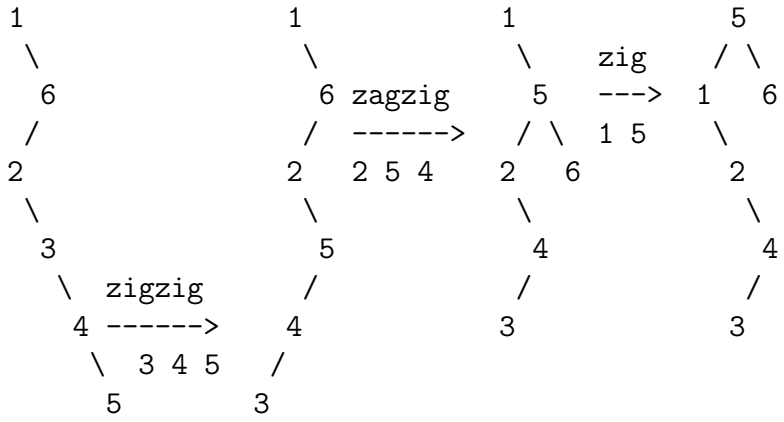
Lösningsförslag till tentamen för Datastrukturer DV

DAG : 25 maj 2009

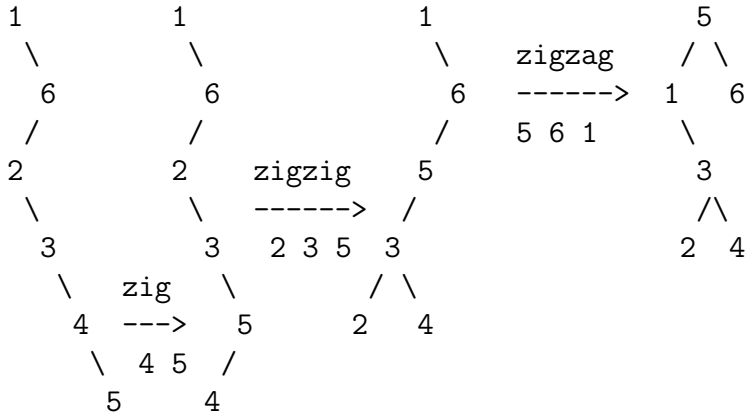
Del A

Datastrukturer på abstrakt nivå.

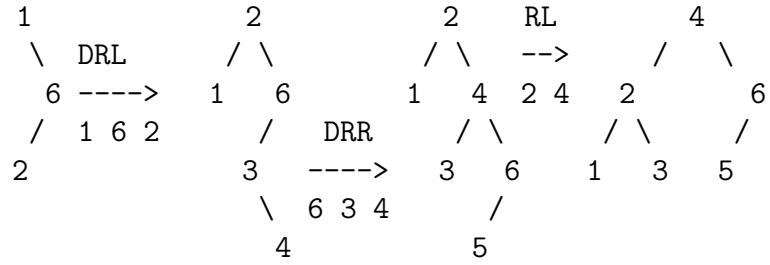
Uppg 1: **a)**



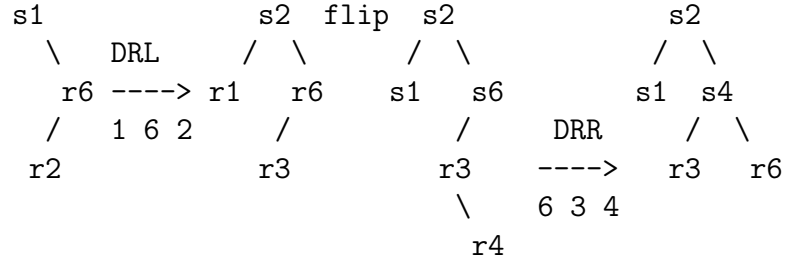
alternativt

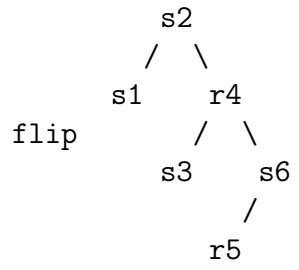


b)


1
2
2
RL
4
DRL
6 ----> 1 6
1 6 2
DRR
---->
1 4
2 4
2
6
2
3
3 6
1 3 5
6 3 4
4
5

c)


s1
s2
flip
s2
s2
DRL
r6 ----> r1 r6
s1 s6
s1 s4
1 6 2
DRR
r2
r3
r3
----> r3 r6
6 3 4
r4


s2
s1
r4
flip
s3
s6
r5

Uppg 2: Vilken eller vilka av följande påstående är sann(a) och vilken eller vilka är falsk(a). Svara bara med Sant eller Falskt. För varje delfråga ger rätt svar 1 p och fel svar ger -0.6 p. Poängen för hela uppgiften kan däremot inte bli negativ.

a) Falskt, en kö har nomalt $\mathbf{O}(\,1\,)$ medan en prioritetskö har $\mathbf{O}(\,^2log\,n\,)$

b) Falskt, för att ta bort sista elementet måste jag ha tag på det näst sista elementet.

c) Falskt, det beror på hur hash-funtionen ser ut eller på hur många kollisioner som uppstått.

d) Falskt. Genom att använda någon form av adresssortering (t.ex. 'bucket sort') till ett fält av storleken 1001 element och sedan gå genom och samla upp elementen fås

$$max(1000, n) \;\in\; \mathbf{0}(n)$$

e) Sant, se föreläsnings-OH.

f) Sant, samma arbete utförs alltid.

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.

0	1	2	3	4	5	6	7	8	9	10	11
80	90	110	40	30	60	100	50	120	10	20	70
6,9		11				p0		8	0	10	2
10	90	70	40	30	60	80	50	20	100	120	110
4,2	8	4,0		p0,2				1	-	11	p10
10	20	30	40	70	60	80	50	90	100	110	120
1,1	p0,0	-	6,7			p3	3		-	-	
10	20	30	50	70	60	40	80	90	100	110	120
	-	-	5,5	6	p3,3	4	-		-	-	-
10	20	30	50	40	60	70	80	90	100	110	120
-	-	-	4	p3	-		-	-	-	-	-
10	20	30	40	50	60	70	80	90	100	110	120
-	-	-	-		-	-	-	-	-	-	-

Uppg 4: **a)** För att lösa problemet behöver vi hitta alla noder vars kortaste väg från startnoden är kortare eller lika med den angivna totalvikten. Vi använder därför en variant Dijkstra's algoritm för kortaste vägen och sparar alla noder vars kortaste väg är mindre än eller lika med den angivna totalvikten. (Alternativt undersöka alla vägar enligt djupet först, se nästa sida)

Vi behöver en prioritetskö, som ordnar par av $< nod, vägvikt >$ efter $vägvikt$.

Vi startar med en tom prioritetskö que
och en tom lista res.

Lägg $< startnod, 0 >$ i que.
så länge kön ej är tom
    låt $< n, vv >$ vara första element i que
    om n ej är markerad
        markera n
        lägg n i res
        för varje grannbåge $< n, x, v >$ till n
            om grannen x ej är markerad
                och $v + vv \leq totvikt$
                    lägg $< x, v + vv >$ i que.
    tag bort första elementet i que
resulatet finns nu i res.

b) Om totalvikten är tillräckligt stor, så kommer alla bågar medföra en inläggning och ett uttag ur kön. Dvs om $|E|$ är antalet bågar, är komplexiteten av $\mathbf{O(}\ |E| * {}^2log\,|E|\ \mathbf{)}$

a) Vi kan enligt den s.k. kallade 'trial and error' metoden söka oss in i grafen enligt djupet först och spara alla noder som passeras i en mängd. För att göra detta användes lämpligen en rekursiv metod, som avbrytes när vägen tar slut (dvs blir för lång). Effektiviten blir dock lidande av detta,

Skapa ett fält *besökt* med alla noder obesökta.
Låt *max* vara den maximala totala väglängden.
Skapa en tom mängd *res* där resultatet skall lagras.
Anropa metoden *sök*(startnode, 0.0)
Resultatet finns nu i *res*

```
sök( nod, vikt )
    markera nod besökt i besökt
    lägg till nod i res
    för alla grannbågar från nod
        om vikt + bågens vikt <= max
            och bågens tillnod ej besökt
                anropa rekursivt
                sök( bågens tillnod, vikt + bågens vikt )
    markera nod ej besökt i besökt
```

b) Komplexiteten blir i detta fall ganska hög. Om parallella bågar och bågar till noden själv **ej** är tillåtna blir komplexiteten $\mathbf{O}(\ |V|^2\ )$.
Annars blir den ungefär $\mathbf{O}(\ \frac{|E|^2}{|V|}\ )$
$|E|$ är antalet bågar och $|V|$ är antalet noder.

Uppg 5: **a)**

```
public boolean add(E elem) {
   if (last == null) {
      last = new Entry(null, null, elem );
      last.next = last.previous = last
   }
   else {
      last = last.next =
          new Entry( last.next, last, elem );
      last.next.previous = last;
   }
   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 remove( Object e ) {
   boolean found = false;
   if ( last != null ) {
      Entry p = last.next;
      if ( p.elem.equals(e) )
         found = true;
      else {
         p = p.next;
         while ( ! found && p != last.next  )
            if ( p.elem.equals(e) )
               found = true;
            else
               p = p.next;
      }
      if ( found )
        removeThis(p);
   }
   return found;
}  // remove

protected void removeThis( Entry p ) {
   if ( p.next == p )
      last = null;
   else {
      p.next.previous = p.previous;
      p.previous.next = p.next;
      if ( p == last )
         last = p.previous;
   }
} // removeThis
```

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

`remove` måste först leta upp den nod som skall tas bort, sedan anropas `removeThis`. `removeThis` arbetar lokalt med `previous` och `next`, dvs säga $\mathbf{O}(1)$ och därför blir komplexiteten för $\mathbf{O}(n)$ för `remove`.

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

Uppg 6:

```
public Iterator<Edge> minSpanningTree( int startNode ) {

  PriorityQueues<Edge> pq  = new PriQueHeap<Edge>();
  Collection<Edge>     res = new LinkedCollection<Edge>();
  boolean[]            inc = new boolean[graph.length];

  for( int j = 0; j < graph.length; j++ )
    if ( graph[startNode][j] > -1 )
      pq.add( new Edge( startNode, j, graph[startNode][j] ));
  inc[startNode] = true;
  int maxNoOfEdges = graph.length - 1;

  while ( ! pq.isEmpty() && res.size() < maxNoOfEdges ) {
    Edge e = pq.removeMin();
    if( ! inc[e.node2] ) {
      res.add(e);
      inc[e.node2] = true;
      for( int j = 0; j < graph.length; j++ )
        if ( graph[e.node2][j] > -1 )
          pq.add( new Edge( e.node2, j, graph[e.node2][j] ));
    }
  }
  return res.iterator();
}
```

Uppg 7: Antingen kan man använda en stack:

```
import BinTree
import Stack

preorder :: BinTree a -> [a]
preorder t = preo  $  push t emptyStack

preo :: Stack ( BT a ) -> [a]
preo st
 | Stack.isEmpty st         = []
 | BinTree.isEmpty (top st) = preo (pop st)
 | otherwise
   = rootVal t : preo(push(leftSub t)( push(rightSub t)(pop st)))
     where t = top st
```

eller, kanske elegantare kan man använda högre ordningens funktioner

```
import BinTree

preorder :: BinTree a -> [a]
preorder t = preo t []

preorder :: BinTree a -> [a] -> [a]
preo t as
 | isEmpty t  = as
 | otherwise
   = rootVal t : preo (leftSub t) ( preo (rightSub t) as )
```