Institutionen för
Datavetenskap
Göteborgs universitet

VT10
DIT960
17/8 -10

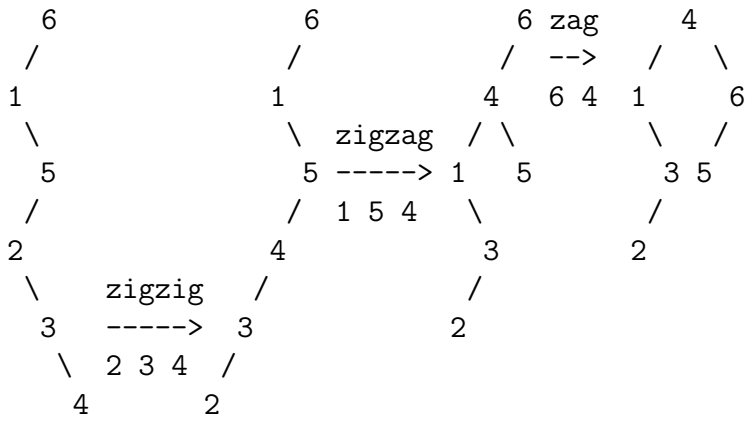
Lösningsförslag till tentamen för Datastrukturer DV

DAG : 19 augusti 2010

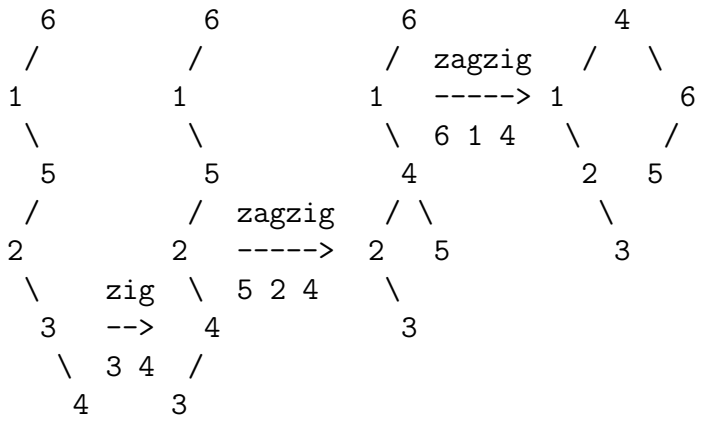
Del A

Datastrukturer på abstrakt nivå.

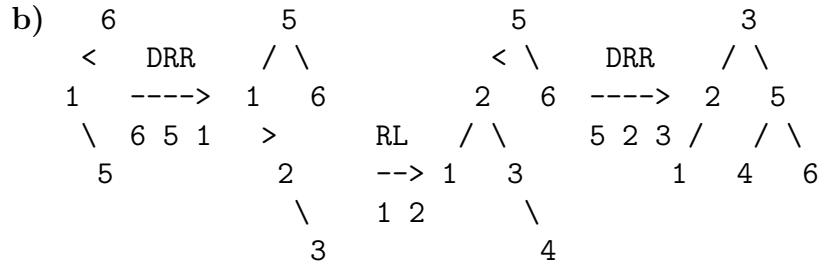
Uppg 1: a)



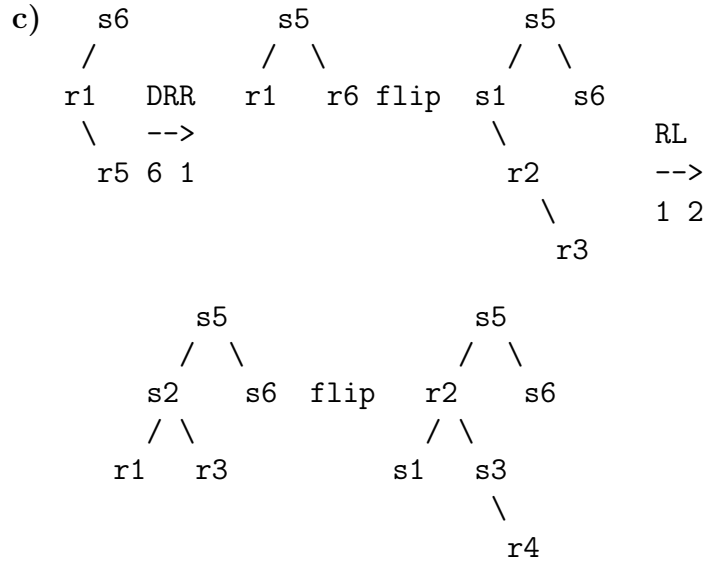
alternativt



b)


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

c)


s6
s5
s5
/ / \ / \
r1 DRR r1 r6 flip s1 s6
\ --> \ RL
r5 6 1 r2 -->
\ 1 2
r3
s5
s5
/ \ / \
s2 s6 flip r2 s6
/ \ / \
r1 r3 s1 s3
\
r4

Uppg 2: **a)** Sant. Visserligen kan det ena delträdet vara upp till dubbelt så högt som det andra, men det påverkar inte komplexiteten (en faktor två stryks ju i komlexitetsuttrycket).

b) Falskt, vi kan ju i Haskell inte ha någon referens till det sista elementet i kön.

c) Sant. Komplexíteten är $\mathbf{O}(\,^2log\, n\,)$

d) Falskt. Det blir exakt samma träd endast om alla vikter är olika. Annars beror det på i vilken ordning bågarna kommer ur prioritetskön. Eftersom prioritetsköerna är olika stora så är det ju lite slumpmässigt vilken båge som kommer först vid lika vikt.

e) Falskt. Det första elementet tas enkelt bort, men för det sista elementet måste vi gå igenom hela listan för att ändra referensen i det näst sista elementet.

f) Sant. Komplexiteten är alltid $\mathbf{O}(\, n * {}^2log\, n\,)$, varken mer eller mindre.

Uppg 3: Vi behöver två löpvariabler (i,j) och en extra variabel (t), vars värden jag skriver ut efter fältet.

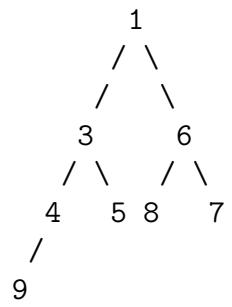
0	*1*	*2*	*3*	*4*	*5*	i	j	t
20	10	50	30	60	40	0	-	-
20	20	50	30	60	40	1	1	10
10	20	50	30	60	40	1	0	10
10	20	50	50	60	40	3	3	30
10	20	30	50	60	40	3	2	30
10	20	30	50	60	60	5	5	40
10	20	30	50	50	60	5	4	40
10	20	30	40	50	60	5	3	40

b) Bästafallet blir vid redan sorterat fält och är av $\mathbf{O}(\, n\,)$

Värstafallet blir vid omvänt sorterat fält och är av $\mathbf{O}(\, n^2\,)$, eftersom variabeln j får gå hela vägen ner till 0 för varje i.

Genomsnittsfallet blir forfarande av $\mathbf{O}(\, n^2\,)$, eftersom variabeln j i genomsnitt får gå halva vägen ner till 0 för varje i, vilket bara motsvarar en konstant i komplexitetsuttrycket.

Uppg 4: **a)**



b) Värdet 6 som ligger under index 7 kan inte vara mindre än värdet av fadern som ligger under $7/2 = 3$, som är 7.

c)

b:	1	2	6	3	5	8	7	9	4
index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

d)

b:	2	3	6	4	5	8	7	9
index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

Uppg 5:

```
private class DGIterator
        implements Iterator<Integer> {

  private Queues<Integer> nextInFront;
  private boolean[]           visited;

  public DGIterator( int startNode ) {
    nextInFront = new LinkedQueue<Integer>();
    visited     = new boolean[ neighbours.length ];
    nextInFront.enqueue( startNode );
    visited[ startNode ] = true;
  } // constructor DGIterator

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

  public Integer next() {
    Integer next = nextInFront.dequeue();
      // if queue empty, right exception is thrown

    for ( E e : neighbours[ next ] )
      if ( ! visited[ e.to ] ) {
        nextInFront.enqueue( e.to );
        visited[ e.to] = true;
      }
    return next;
  } // next

  public void remove() {
    throw new UnsupportedOperationException();
  } // remove
} // class DGIterator
```

Uppg 6: **a)**

```
import BinSearchTree

height :: BST a -> Int
height at
 | isEmpty at = 0
 | otherwise  = 1 + max (height (leftSub at))
                        (height (rightSub at))
```

b)

```
import BinSearchTree

preOrder :: BST a -> [a]
preOrder at
 | isEmpty at = []
 | otherwise  = preo at []

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

Uppg 7:

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

  private Stacks<TreeNode<E>> nextOnTop;
  TreeNode<E>         lastNext;

  public BinTreeIterator() {
    nextOnTop  = new LinkedStack<TreeNode<E>>();
    lastNext   = null;
    pushLeftChain(root);
  }  // constructor BinTreeIterator

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

  public E next() {
    lastNext = nextOnTop.pop();
    pushLeftChain(lastNext.right);
    return lastNext.element;
  } // next

  public void remove() {
    if (lastNext == null)
      throw new IllegalStateException();

    // Note that in this way, the stack in an
    // iterator does not need to be changed
    if ( lastNext.left == null )
      liftRightSubtree();
    else
      swapWithRightMostInLeftTree();
    lastNext = null;
  }  // remove
```

Hjälpmetoderna `pushLeftChain`, `liftRightSubtree` och `swapWithRightMostInLeftTree` på nästa sida.

```
  private void pushLeftChain( TreeNode<E> p ) {
    for( TreeNode<E> q = p; q != null; q = q.left )
      nextOnTop.push( q );
  } // pushLeftChain

  private void liftRightSubtree() {
    if ( lastNext.right != null )
      lastNext.right.parent = lastNext.parent;
    if ( lastNext.parent == null )
       root = lastNext.right;
    else if ( lastNext.parent.left == lastNext )
       lastNext.parent.left = lastNext.right;
    else
      lastNext.parent.right = lastNext.right;
  } // liftRightSubtree

  private void swapWithRightMostInLeftTree() {
    if ( lastNext.left.right == null ) {
      lastNext.element = lastNext.left.element;
      lastNext.left = lastNext.left.left;
      if ( lastNext.left != null )
        lastNext.left.parent = lastNext;
    }
    else {
      TreeNode<E> p = lastNext.left;
      while ( p.right.right != null )
        p = p.right;
      lastNext.element = p.right.element;
      p.right = p.right.left;
      if ( p.right != null )
        p.right.parent = p;
    }
  } // swapWithRightMostInLeftTree

} // class BinTreeIterator
```