Institutionen för
Datavetenskap
Chalmers TH, Göteborgs universitet

VT09
TDA416, DIT721
09-03-13

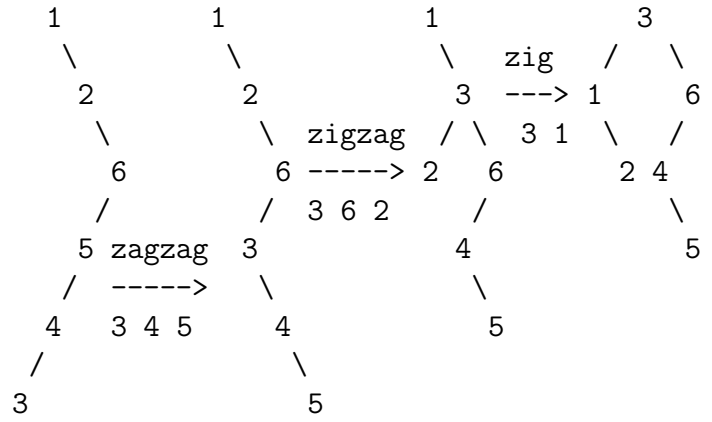
Lösningsförslag till tentamen för Datastrukturer och algoritmer

DAG : 13 mars 2009

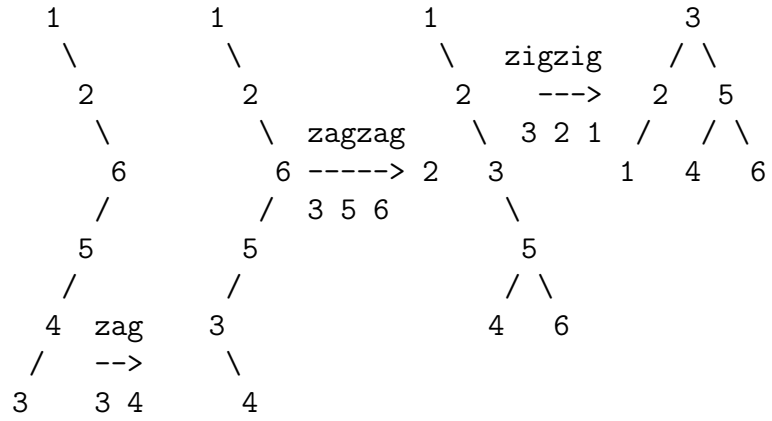
Del A

Datastrukturer på abstrakt nivå.

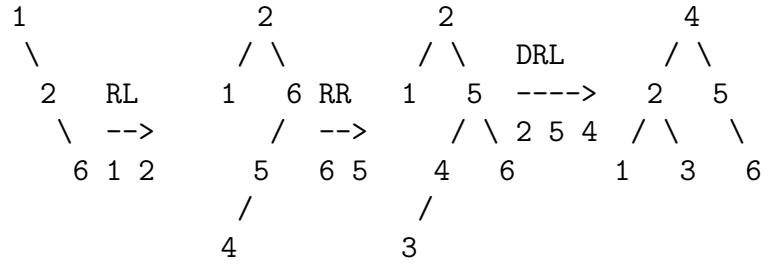
Uppg 1: a)



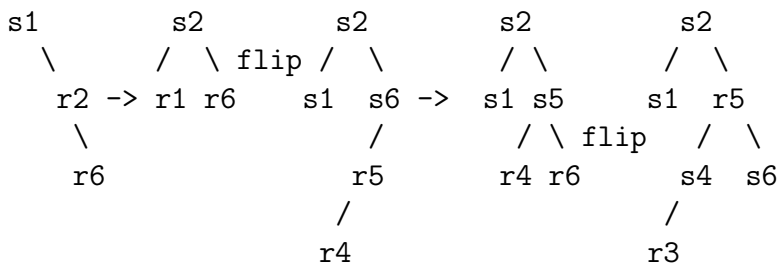
eller



b)


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

c)


s1
s2
r2 -> r1 r6
r6
s2
flip
s1
s6 ->
r5
r4
s2
s1 s5
r4 r6
s2
flip
s1
r5
s4
s6
r3

Uppg 2: **a)** Falskt. Urvalssortering gör alltid samma mängd av jobb.

b) Falskt. Att gå till mitten av en länkad lista är alldeles för dyrt.

c) Sant. Genom att ha en referens till sista elementet, kan man nå både första och sista elementet med komplexiteten $\mathbf{O}(\,1\,)$

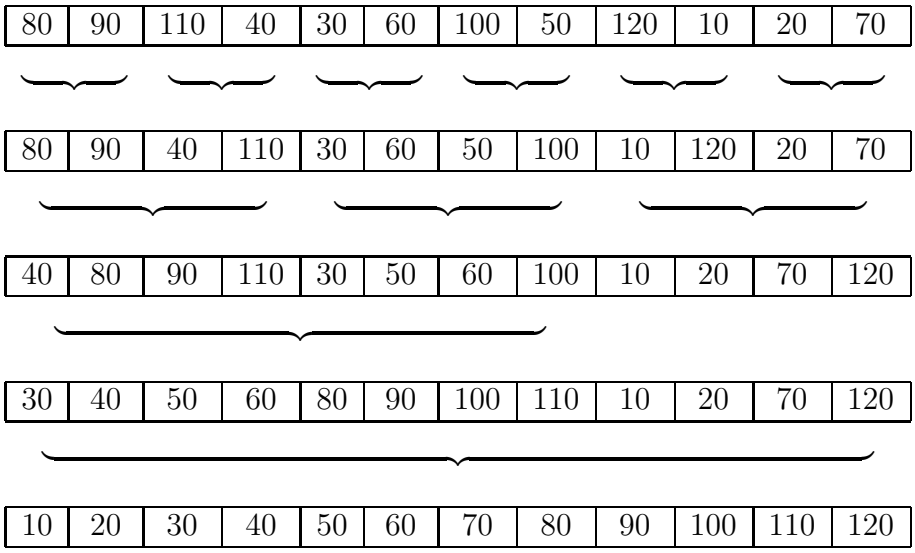
d) Falskt. Det kan dels bero på antalet kollisioner och dels på hash-funktionens komplexitet.

e) Falskt. Den är $\mathbf{O}(\,n^2\,)$.

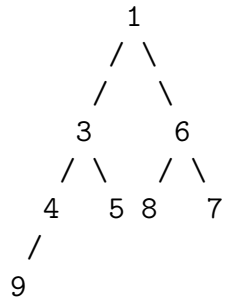
f) Sant. Eftersom sammanhängande betyder att det finns en väg för varje nod till alla andra noder.

g) Falskt. Eftersom ett nätverk är en riktad graf, för vilken Prims algoritm ej fungerar.

Uppg 3: Jag väljer här det iterativa tankesättet. Dvs först har vi 'delfiler' av storleken 1, och gör alltså merge parvis. Sedan sorleken 2 osv.



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*

Del B

Datastrukturer på implementeringsnivå.

Uppg 5: **a)**

```
public List<Integer> traversBF( int startnode ) {

   boolean[] visited
       = new boolean[neighbours.length];
   List<Integer> res
       = new LinkedList<Integer>();
   Queues<Integer> que
       = new LinkedQueue<Integer>();

   Arrays.fill( visited, false);
   que.enqueue( startnode );
   visited[ startnode ] = true;
   res.add( startnode );

   while ( ! que.isEmpty() ){
      for( E e : neighbours[ que.dequeue() ] )
         if ( ! visited[ e.to ] ) {
            que.add( e.to );
            visited[ e.to ] = true;
            res.add( e.to );
         }
   }
   return res;
} // traversBF
```

b) Eftersom varje båges `to`-värde läggs i kön högst 1 gång, så är komplexiteten högst $\mathbf{O}(|E|)$. Och, eftersom maximala antalet bågar är $n * (n-1)$, så är komplexiteten högst $\mathbf{O}(|V|^2)$.

Uppg 6: **a)**

```
public int noOfLeafs() {
  return noOfLeafs( root );
}

public int noOfLeafs(TreeNode<E> t) {
  if ( t == null )
     return 0;
  else if ( t.left == null && t.right == null )
     return 1;
  else
     return noOfLeafs( t.left ) +
            noOfLeafs( t.right );
}
```

b)

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

  protected Stacks<TreeNode<E>> nextOnTop;

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

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

    public E next() {
      TreeNode<E> lastNext = nextOnTop.pop();
        // Notera att om stacken är tom så
        // kastas rätt typ av exception,
        // som är ett RunTimeException varför
        //  någon extra kodinte behövs !
      if ( lastNext.right != null )
        nextOnTop.push(lastNext.right);
      if ( lastNext.left != null )
        nextOnTop.push(lastNext.left);
      return lastNext.element;
    } // next

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

c) Till att börja med måste vi spara den senast givna noden, så att vi vet vilken nod som skall bort. vid `remove`. Detta görs genom att skriva över `remove`-metoden.

Vidare behöver vi leta upp fadernoden till den nod som skall bort vilket vi gör i metoden `fatherOf`

Om node som skall bort i `remove` ej har ett vänster delträd, kan vi helt enkelt länka förbi den i fadernoden. Annars kan vi göra på två sätt:

1. Flytta upp hela vänsterkedjan ett steg, och vrid in det sista högerträdet i vänsterkedjan längst till vänster.

2 Flytta höger delträd, av noden som skall bort, och lägg det längst till höger i vänster delträd. Länka sedan förbi noden som skall bort.

Båda metoderna kräver att man också justerar stacken något, vilket många hade glömt på tentan.

```
private class BinTreeIteratorWithRemove
        extends BinTreeIterator {

  TreeNode<E> lastNext;

  public E next() {
   lastNext = nextOnTop.top();
           return super.next();
  } // next

  private TreeNode<E> fatherOf( TreeNode<E> curr ) {
    // Får ej anropas om fadern är roten !!
    // Går nog snabbast att hitta med bredden först
    Queues<TreeNode<E>> que = new LinkedQueue<TreeNode<E>>();
    TreeNode<E> father = root;
    while ( father.left != curr && father.right != curr ) {
      if ( father.left != null )
        que.enqueue( father.left );
      if ( father.right != null )
        que.enqueue( father.right );
      father = que.dequeue();
    }
    return father;
  } // fatherOf
```

På tentan hade de flesta valt den andra metoden, själv hade jag valt den första (vilket 2 hade gjort på tentan), eftersom trädet i metod 2 blir lätt mycket skevt och efter några urtagningar börjar likna en lista. Här presenteres därför först metod 1.

```
public void remove() {

  if ( lastNext == null )
    throw new IllegalStateException();

  if ( lastNext.left == null )
    // Vi behöver då hitta fadernoden och
    // länka förbi noden som skall tas bort.
    // Går nog snabbast med bredden först
    if ( lastNext == root )
      root = root.right;
        // Notera att denna nod måste ligga på
        // toppen av stacken !
    else {
       TreeNode<E> father = fatherOf( lastNext );
       if ( father.left == lastNext )
          father.left = lastNext.right;
       else
          father.right = lastNext.right;
       }
     // Fortsättning nästa sida
```

```
    else { //   lastNext != null
      // Flytta elementen i 'hela vänsterkedjan'
      // ett steg uppåt och lägg det sista
      // högerträdet längst ner till vänster.

      TreeNode<E> p = lastNext;
      p.element = p.left.element;
      while (p.left.left != null) {
         p = p.left;
         p.element = p.left.element;
      }
      p.left = p.left.right;

       // slutligen måste stacken justeras,
       // eftersom nästa element har bytt
       // plats till lastNext
      nextOnTop.pop();
      nextOnTop.push(lastNext);
    }
    lastNext = null;
 } //remove
} // BinTreeIteratorWitwRemove
```

Alternativ else-del enlig metod 2.

```
    else { //   lastNext.left != null
       if ( lastNext.right != null ) {
           // flytta höger delträd längst
           // till höger i vänster delträd
         rightMostOf( lastNext.left ).right = lastNext.right;
           // Nu måste lastNext.right tas bort ur stacken !!
         nextOnTop.pop();
         nextOnTop.pop();
         nextOnTop.push( lastNext.left );
       }
         // länka förbi noden som skall bort
       if (lastNext == root)
         root = root.left;
       else {
         TreeNode<E> father = fatherOf( lastNext );
         if ( father.left == lastNext )
           father.left = lastNext.left;
         else
           father.right = lastNext.left;
       }
    }
  }
  lastNext = null;
 } //remove
} // BinTreeIteratorWithRemove
```

Och vi behöver allså metoden:

```
   private TreeNode<E> rightMostOf( TreeNode<E> t ) {
       if (t.right == null)
   return t;
       else
   return rightMostOf( t.right );
   } // rightMostOf
```

Slutligen ett litet testprogram, som inte ingick i tentan

```
public static void main( String[] args ) {
  BinTree<Integer> t = new BinTree<Integer>();
  t.root = new TreeNode<Integer> (
              new TreeNode<Integer>(
          new TreeNode<Integer>(1),
                  2,
                  new TreeNode<Integer>(
                      new TreeNode<Integer>(3),
                      4,
                      null
                  )
               ),
               5,
               new TreeNode<Integer>(
                   null,
                   6,
                   new TreeNode<Integer>(7)
               )
           );

  System.out.println( t.noOfLeafs() );
  for ( int n : t )
    System.out.print( " " + n );
    System.out.println();
    Iterator<Integer> it = t.iteratorWithRemove();
    it.next(); it.remove();  // dvs ta bort 5
    it.next(); it.next(); it.next();
    it.remove(); // dvs ta bort 3
    for ( int n : t )
      System.out.print( " " + n );
    System.out.println();
} // main

/*
sed:bjerner:[~/itds/tentor/program]$ java BinTree
3
 5 2 1 4 3 6 7
 2 1 4 6 7
 */
```