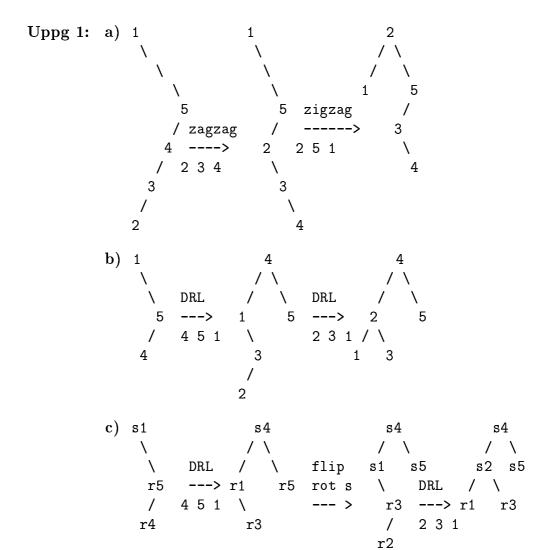
Institutionen för Datavetenskap Chalmers TH, Göteborgs universitet $VT08 \\ TDA416 (TDA415), \ DIT721 \ (INL080) \\ 08-03-13$

Lösningsförslag till tentamen för Datastrukturer och algoritmer

DAG: 15 mars 2008

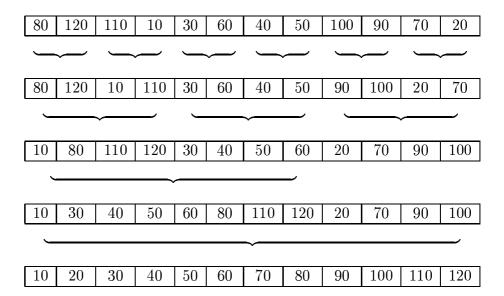
Del A

Datastrukturer på abstrakt nivå.



- **Uppg 2:** a) Falskt. Urvalssorteringen utför alltid samma antal operationer, dvs $\mathbf{0}(n^2)$ jämförelser. (och $\mathbf{0}(n)$ antal flyttningar)
 - b) Falskt. Eftersom det tar $\mathbf{0}(n)$ steg att ta sig till slutet av listan vilket måste göras antingen när man lägger till eller tar bort ett element i kön. Däremot kan man ha en referens till sista elementet och då kan vi ordna det i $\mathbf{0}(1)$.
 - c) Sant. Båda har en komplexitetet av $\mathbf{0}(n^2)$.
 - d) Falskt. Insättningssortering har $\mathbf{0}(n)$ medans snabbsortering har $\mathbf{0}(^2 \log n)$ som bästafallskomplexitet.
 - e) Sant. Om trädet inte redan är binärt, låt första sonen vara rot i höger delträd och bind ihop syskonen genom att andra sonen blir rot i höger delträd till första sonen, tredje sonen blir rot i höger delträd till andra sonen osv.
 - f) 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.
 - g) Falskt, du måste fortfarande gå igenom listan för att nå sista elemenetet. (Du måste också ha en refererens till sista elentet.)
 - h) Falskt. Med en dålig hashning kan komplexiteten vara ända upp till O(n). Detta händer om t.ex. alla nycklarna ger samma hashkod.

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



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 $< nod, v \ddot{a}gvikt >$ efter $v \ddot{a}gvikt$.

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 \le 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 **O(** $|E| * {}^2 log |E|$)

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 $bes\"{o}kt$ med alla noder obes\"{o}kta. Låt max vara den maximala totala väglängden. Skapa en tom mängd res där resultatet skall lagras. Anropa metoden $s\"{o}k$ (startnode, 0.0) Resultatet finns nu i res

```
s\ddot{o}k(\ nod,\ vikt\ ) markera nod besökt i bes\ddot{o}kt lägg till nod i res för alla grannbågar från nod om vikt\ +\ bågens\ vikt\ j=\ max och bågens tillnod ej besökt anropa rekursivt s\ddot{o}k(\ bågens\ tillnod,\ vikt\ +\ bågens\ vikt\ ) markera nod ej besökt i bes\ddot{o}kt
```

b) Komplexiteten blir i detta fall ganska hög. Om parallella bågar och bågar till noden själv ej är tillåtna blir komplexiteten $O(|V|^2)$.

Annars blir den ungefär O(|E| * |E| / |V|) |E| är antalet bågar och |V| är antalet noder.

Del B

Datastrukturer på implementeringsnivå.

```
Uppg 5: a) public int noOfUnaryNodes() {
    return noOfUnaryNodes(root);
} // noOfUnaryNodes

private int noOfUnaryNodes(TreeNode<E> t) {
    if ( t == null ||
        t.left == null && t.right == null )
    return 0;
else if ( t.left != null && t.right != null )
    return noOfUnaryNodes(t.left) +
        noOfUnaryNodes(t.right);
else
    return 1 + noOfUnaryNodes(t.left) +
        noOfUnaryNodes(t.right);
} // noOfUnaryNodes
```

b) noOfUnaryNodes anropar endast rekursiva noOfUnaryNodes och får alltså dess komplexitet.

Rekursiva no<code>OfUnaryNodes</code> anropas en gång per nod i trädet samt en gång för varje tomt träd. Antalet tomma träd är emellertid n+1, där n är antalet noder. DVS komplexiteten blir $\mathbf{0}(n)$.

```
c) private class BinTreeIterator
           implements Iterator<E> {
     private Queues<TreeNode<E>> nextInFront;
     public BinTreeIterator() {
       nextInFront =
          new LinkedQueue<TreeNode<E>>();
       if ( root != null )
         nextInFront.enqueue(root);
     } // constructor BinTreeIterator
     public boolean hasNext() {
       return ! nextInFront.isEmpty();
     } // hasNext
     public E next() {
          TreeNode<E> t = nextInFront.dequeue();
            // Notera att om stacken är tom så
            // kastas rätt typ av exception, som
            // är ettRunTimeException varför
            // någon extra kod inte behövs !
          if ( t.left != null )
             nextInFront.enqueue(t.left);
          if ( t.right != null )
             nextInFront.enqueue(t.right);
          return t.element;
     } // next
     public void remove() {
        throw new UnsupportedOperationException();
     } // remove
   } // class BinTreeIterator
```

Uppg 6: Denna metod kan definieras antingen med en stack eller medelst rekursion (som ju är en implicit stack!). Vi börjar med det första alternativet.

```
public List<Integer> traversDF(int startnode){
  boolean[] visited
      = new boolean[neighbours.length];
  Arrays.fill( visited, false);
  List<Integer> res
      = new LinkedList<Integer>();
  Stacks<Integer> stack
      = new LinkedStack<Integer>();
  stack.push( startnode );
  while ( ! stack.isEmpty() ){
    int node = stack.pop();
    if ( ! visited[node] ) {
      res.add( node );
      visited[ node ] = true;
      for ( Edge e : neighbours[ node ] )
        if ( ! visited[e.to] )
          stack.push(e.to);
  }
  return res;
} // traversDF
```

Medelst rekursion får vi istället:

b) Blir samma sak som ovan!

```
a) public List<Integer> traversDF(int startnode){
     boolean[] visited
         = new boolean[neighbours.length];
     Arrays.fill( visited, false);
     List<Integer> res
         = new LinkedList<Integer>();
     visitNode( startnode, visited, res );
     return res;
   } // traversDF
   private void visitNode( int node,
                           boolean[] visited,
                           List<Integer> res ) {
     visited[node] = true;
     res.add( node );
     for ( Edge e : neighbours[ node ] )
       if ( ! visited[e.to] )
         visitNode( e.to, visited, res);
   } // visitNode
```

Uppg 7: Först behöver vi en jämförbar klass att stoppa i prioritetskön:

```
class NodePathWeight
implements Comparable<NodePathWeight> {
   int node;
   double pathWeight;
   NodePathWeight( int node, double pathWeight ) {
      this.node
                      = node;
      this.pathWeight = pathWeight;
   } // constructor NodeWeight
   public int compareTo( NodePathWeight npw ) {
      double jfr = pathWeight - npw.pathWeight;
      return jfr < 0 ? -1 : jfr > 0 ? 1 : 0;
   } // compareTo
} // class NodeWeight
Sen är vi färdiga att implementera metoden:
public Iterator<Integer>
       reachable( int startNode, double totalWeight ) {
  boolean[] visited = new boolean[neighbours.length];
  List<Integer> res = new LinkedList<Integer>();
  PriorityQueues<NodePathWeight> que
     = new PriQueHeap<NodePathWeight>();
  Arrays.fill( visited, false);
  que.add( new NodePathWeight( startNode, 0 ));
  while ( ! que.isEmpty() ) {
    NodePathWeight npw = que.removeMin();
    if ( ! visited[npw.node] ) {
      visited[npw.node] = true;
      res.add( npw.node);
      for ( E e : neighbours[ npw.node ] ) {
        double pathWeight = e.weight() + npw.pathWeight;
        if ( ! visited[e.to] && pathWeight <= totalWeight )</pre>
          que.add( new NodePathWeight(e.to, pathWeight ));
  } } }
  return res.iterator();
} // reachable
```

```
public Iterator<Integer>
       reachable( int startNode, double maxTotalWeight ) {
  boolean[]
               visited = new boolean[neighbours.length];
  Set<Integer> res
                       = new HashSet<Integer>();
  Arrays.fill( visited, false);
  search( startNode, 0.0, maxTotalWeight, visited, res );
  return res.iterator();
} // reachable
private void search( int
                                   node,
                     double
                                   totalWeight,
                     double
                                   maxTotalWeight,
                     boolean[]
                                   visited,
                                            ) {
                     Set<Integer> res
   res.add(node);
   visited[node] = true;
   for ( E e : neighbours[node] )
      if ( ! visited[e.to] &&
           e.weight() + totalWeight <= maxTotalWeight )</pre>
         search( e.to,
                 e.weight() + totalWeight,
                 maxTotalWeight,
                 visited,
                                             );
                 res
   visited[node] = false;
} // search
```