Lösningsförslag till tentamen

Datastrukturer, DAT037 (DAT036), 2017-01-11

1. Loopen upprepas `n` gånger.

   `getAt` på en dynamisk array tar tiden $O(1)$.

   `member` på ett AVL-träd av storlek `n` tar tiden $O(\log \mathtt{n})$.

   `addLast` på en dynamisk array tar tiden $O(1)$ amorterat, `n` anrop av metoden tar tiden $O(\mathtt{n})$.

   Tiden det tar att utföra en iteration av loopen är oberoende av värdet på `i`.

   $$T(\mathtt{n}) = O(\mathtt{n}(1 + \log \mathtt{n}) + \mathtt{n}) = O(\mathtt{n} \log \mathtt{n})$$

2. i)
```
public int height() {
  return nodeHeight(root);
}
private int nodeHeight(Node node) {
  if (node == null) return -1;
  return 1 + Math.max(nodeHeight(node.left),
                      nodeHeight(node.right));
}
```

   Tidskomplexitet: `height` tar konstant tid bortsett från anropet till `nodeHeight`. `nodeHeight` tar i värsta fall konstant tid bortsett från rekursiva anropen. `nodeHeight` anropas en gång för roten och två gånger för varje nod. Antalet noder är n. $T(n) = O(1 + (1 + 2n) \cdot 1) = O(n)$

   ii)
```
public boolean isAVLBalanced() {
  return balancedHeight(root) >= -1;
}
// Returnerar höjden om delträdet är balanserat
// annars -2
private int balancedHeight(Node node) {
  if (node == null) return -1;
  int hl = balancedHeight(node.left);
  if (hl == -2) return -2;
  int hr = balancedHeight(node.right);
  if (hr == -2) return -2;
  if (Math.abs(hl - hr) <= 1)
    return 1 + max(hl, hr);
  else
    return -2;
}
```

   Tidskomplexitet: Samma resonemang som för i).

3. Ett lösning är att lägga alla startnoder i kön från början.

   Använd en grannlista för att lagra grafen. Använd avbildning implementerad med en Hashtabell och lista implementerad med dynamisk array. Definiera en hjälpklass

```
private class Edge {
  int weight;
  int adjNode;
}
```

   och använd föjande fält:

```
HashMap<Integer, ArrayList<Edge>> a;
```

   Implementering av `addNode`:

```
addNode(i)
  a.put(i, new ArrayList<Edge>)
```

   Skapa en ny dynamisk array tar konstant tid och lägga in element i hashtabell tar amorterad konstant tid. Alltså tidskomplexitet $O(1)$ amorterat.

   Implementering av `addEdge`:

```
addEdge(i, j, w)
  if (!a.containsKey(i)) return
  a.get(i).addLast(ny Edge med adjNode = j, weight = w)
```

   Uppslag i hashtabell och skapa nytt `Edge`-objekt tar konstant tid. Lägga till element i slutet av dynamisk array tar konstant tid amorterat. Tidskomplexitet: $O(1)$ amorterat.

   Implementering av `closestFromNode`: Dijkstras algoritm kan användas. Man kan utföra Dijkstras en gång för varje nod i `f` och välja den med kortast väg. Man kan också utföra Dijkstras en gång genom att lägga in alla noder i `f` i prioritetskön i början. När man hittat slutnoden söker man tillbaka för att hitta startnoden som är närmst. Det andra sättet leder till komplexiteten som efterfrågas för betyg fyra eller fem. Förslag på implementering enligt andra sättet:

```
closestFromNode(f, t)
  let d = new HashMap<Integer, Integer>  // bästa kända avstånd till nod
      v = new HashSet<Integer>  // nod besökt
      p = new HashMap<Integer, Integer>  // förågende nod i kortaste väg
      q = new PriorityQueue<Integer>
           with priority = d.get(i) for each element i
  for each int i in f {
    d.put(i, 0)
    q.offer(i)
  }
```

```
 while q not empty {
  i = q.poll()
  if (i == t) findStartNode (se nedan)
  v.add(i)
  for each Edge e in a.get(i) {
    j = e.adjNode
    if (!v.contains(j)) {
      newd = d.get(i) + e.weight
      if (!d.containsKey(j) || newd < d.get(j)) {
        q.remove(j)
        d.put(j, newd)
        p.put(j, i)
        q.offer(j)
      }
    }
  }
 }
 // t kan ej nås
 return null

findStartNode:
 // t har nåtts, hitta startnod
 i = t
 while (p.containsKey(i)) {
   i = p.get(i)
 }
 return i
```

Initiering av `d`, `v`, `p` och `q` tar konstant tid.

Första loopen upprepas n gånger. Insättning i `d` tar $O(1)$ amorterat. Insättningarna i `q` går på konstant tid amorterat eftersom alla element har samma prioritet. Noderna i `f` är distinkta, så $n \leq v$. Första loopen: $O(n(1+1)) = O(n) = O(v)$

Andra loopen upprepas i värsta fall v gånger eftersom en nod finns med max 1 gång i kön och en besökt nod inte läggs i kön igen. Kön innehåller max v element. Uttag ur `q` tar $O(\log v)$. Insättning i `v` tar $O(1)$ amorterat. Andra loop bortsett från dess inre loop: $O(v \log v)$

Inre loopen i andra loopen upprepas totalt max e gånger. Uppslag i `d` tar $O(1)$. Borttagning ur och insättning i `q` tar $O(\log v)$ (amorterat för insättning). Insättning i `d` och `p` tar $O(1)$ amorterat. Inre loopen totalt: $O(e \log v)$

While-loopen som letar upp startnoden upprepas max v gånger och varje iteration tar konstant tid. $O(v)$

Totalt för operationen: $O(v + v \log v + e \log v + v) = O((v + e) \log v)$

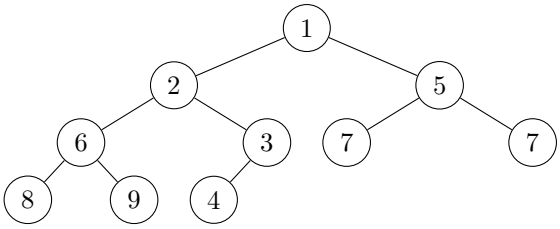
Om man väljer en lösning där element inte uppdateras/tas bort i kön så kan man komma fram till $O(v + e \log e)$.

4. Detta är en topologisk sortering för grafen:

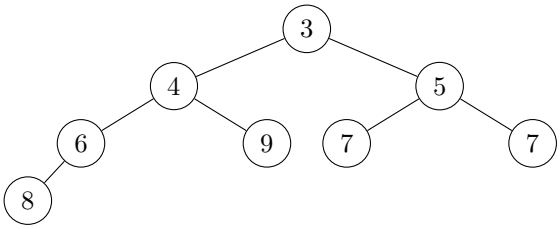
$$6, 10, 9, 8, 5, 7, 4, 2, 1, 3$$

6 och 10 samt 5 och 7 kan byta plats.

5. a) insättning av talet 1



b) anrop av `deleteMin` en gång



6. Lösningsförslag som ger logaritmisk tidskomplexitet för `deleteMinA` och `deleteMinB`. Använd två mängder, `a` och `b`, implementerade med AVL-träd. Låt `a` vara sorterad enligt ordning A och `b` enligt ordning B. Antag att representationen av AVL-träden ser ut så här:

```
public class AVLTree {
    private class Node {
        A    contents;  // Innehåll.
        Node left;      // Vänstra barnet.
        Node right;     // Högra barnet.
        ...  // höjd/balans
    }
    private Node root;  // roten
    ...
}
```

```
empty()
  a = nytt tomt AVL-träd sorterat enligt x.compareTo(y)
  b = nytt tomt AVL-träd sorterat enligt
        (x % 100).compareTo(y % 100)
```

Skapa ett tomt träd tar konstant tid, så $O(1)$.

```
insert(x)
  a.add(x)
  b.add(x)
```

Insättning i AVL-träd tar logaritmisk tid och varje träd innehåller n element, så $O(\log n)$.

```
deleteMinA()
  if (a.isEmpty()) return null;
  x = a.findMin()
  a.remove(x);
  b.remove(x);
  return x;

findMin()
  // trädet ej tomt
  n = root
  while (n.left != null) {
    n = n.left;
  }
  return n.contents;
```

`findMin` är operation hos våra AVL-träd. Exekveringen tar konstant tid förutom loopen. En iteration i loopen tar konstant tid. Antal iterationer är begränsad av trädets höjd och den är begränsad av $\log n$. Tidskomplexiteten för `findMin` är $O(\log n)$.

Tidskomplexiteten för `deleteMinA`: `isEmpty` för träd tar konstant tid. Borttagning ur AVL-träd tar logaritmisk tid. Alltså $O(\log n)$

`deleteMinB` implementeras på samma sätt med enda skillnaden att `b.findMin()` anropas istället.