Lösningsförslag till tentamen

Datastrukturer, DAT037 (DAT036), 2017-08-17

1. Loopen upprepas `n` gånger, för `i` från 0 till $\texttt{n} - 1$. För varje upprepning utförs i värsta fall dessa operationer:

   - 2 stycken `l.getAt`, uppslag i dynamisk array. Det tar $O(1)$.
   - `s.member`, sökning i perfekt hashtabell. Det tar $O(1)$.
   - `t.add`, insättning i AVL-träd som för varje upprepning maximalt har storleken `i`. Det tar $O(\log \texttt{i})$.

   Övrigt i och utanför loopen tar konstant tid.

   Värstafallskomplexiteten blir

   $$T(\texttt{n}) = O(1) + \sum_{i=1}^{\texttt{n}-1}(\log i + 3O(1)) = \sum_{i=1}^{\texttt{n}-1} \log i = O(\texttt{n} \log \texttt{n})$$

2. AD, DE, AB, BC, DF, EH, HG

3. ```
   private void removeAllLessThan(Node n, int x) {
     if (n == null) return;
     if (n.data >= x) {
       removeAllLessThan(n.left, x);
     } else {
       if (n.parent == null) {
         root = n.right;
       } else {
         n.parent.left = n.right;
       }
       if (n.right != null) {
         n.right.parent = n.parent;
         removeAllLessThan(n.right, x);
       }
     }
   }

   public void removeAllLessThan(int x) {
     removeAllLessThan(root, x);
   }
   ```
   ```

4. Antar perfekt hashfunktion för hashtabellerna.

```
class Relation {
  Map<Integer, Set<Integer>> r;

  public Relation() {
    r = new HashMap<>();  // O(1)
  }  // = O(1)
  public void addRelation(int t, int u) {
    if (!r.containsKey(t)) {  // O(1)
      r.put(t, new HashSet<>());  // O(1) + O(1) amorterat
    }
    r.get(t).add(u);  // O(1) + O(1) amorterat
  } // = O(1) amorterat

  public void removeRelation(int t, int u) {
    if (r.containsKey(t)) {  // O(1)
      r.get(t).remove(u);  // O(1) + O(1) amorterat
    }
  } // = O(1) amorterat

  public boolean isRelated(int t, int u) {
    if (!r.containsKey(t)) return false;  // O(1)
    return r.get(t).contains(u);  // O(1) + O(1)
  }  // = O(1)
}
```

5. Antar perfekt hashfunktion för hashtabellen som används. Antar också att iterator för balanserade sökträd som används traverserar trädet inorder, vilket betyder att om två träd innehåller samma nycklar så kommer de besökas i samma ordning.

```
public class Graph {
  private Map<Integer, List<Integer>> alist;
  public Graph() {
    alist = new TreeMap<>();
  }
  public void addNode(int x) {
    alist.put(x, new LinkedList<>());  // O(1) + O(log V)
  }
  public void addEdge(int x, int y) {
    alist.get(x).add(y);  // O(log V) + O(1)
  }
  public boolean equals(Graph g) {
    if (alist.size() != g.alist.size()) return false;  // O(1)
    Iterator<Entry<Integer,List<Integer>>> gait =
        g.alist.entrySet().iterator();  // O(1)
    for (Entry<Integer,List<Integer>> kv : alist.entrySet()) {  // V times
        // implicit next() för for-loopens iterator: O(log V), totalt O(V)
      Entry<Integer,List<Integer>> gkv = gait.next();  // O(log V), totalt O(V)
      if (kv.getKey() != gkv.getKey()) return false;  // O(1)
      List<Integer> a = kv.getValue();  // O(1)
      List<Integer> ga = gkv.getValue();  // O(1)
      if (a.size() != ga.size()) return false; // O(1)
      Set<Integer> gas = new HashSet<>();  // O(1)
      for (int y : ga) {  // max E gånger totalt
          // implicit next() för for-loopens iterator: O(1)
        gas.add(y);  // O(1) amorterat, O(E) totalt
      }
      for (int y : a) {  // max E gånger totalt
          // implicit next() för for-loopens iterator: O(1)
        if (!gas.contains(y)) return false; // O(1)
      }
    }
    return true;
  }
}
```

Lägger man ihop komplexiteten för de olika delarna får man:

- `addNode`: $O(\log V)$
- `addEdge`: $O(\log V)$
- `equals`: $O(V + E)$

6. Antar perfekt hashfunktion.

```
empty() = låt hs vara en mängd implementerad med hashtabell (O(1))
          låt ts vara en mängd implementerad med balanserat sökträd (O(1))

add(x) = sätt in x i hs (O(1)) amorterat
         sätt in x i ts (O(log n))

remove(x) = ta bort x från hs (O(1)) amorterat
            ta bort x från ts (O(log n))

contains(x) = slå upp x i hs (O(1))
```

`successorOf` kan implementeras som en variant på vanligt sökning i sökträd. Antag (förenklat) att noderna i trädet representeras klassen `Node` och att det finns en instansvariabel som pekar på rot-noden.

```
class Node {
  int data;
  Node left, right;
}
Node root;
```

Man kan då implementera metoden så här:

```
public Integer successorOf(int x) {
  return successorOf(x, root);
}

private Integer successorOf(int x, Node n) {
  if (n == null) {
    return null;
  }
  if (x < n.data) {
    Integer res = successorOf(x, n.left);
    if (res == null) res = n.data;
    return res;
  } else {
    return successorOf(x, n.right);
  }
}
```

Liksom för vanlig sökning i sökträd så traverseras en väg. Varje besökt nod tar $O(1)$. Antalet noder i vägen är begränsat av $O(\log n)$ eftersom trädet är balanserat.