DIT960 – Datastrukturer

suggested solutions for exam 2017-08-17

1. Assume that `n` is a non-negative integer, that `s` is an integer set implemented with a hash table and containing at most `n` elements, that `l` is a list of integers implemented with a dynamic array and containing `n` elements, and that `t` is a integer set implemented with an AVL tree. Also assume that `t` is initially empty.

   Under these assumptions, what is the big-O time complexity of the following code?

   ```
   for (int i = 0; i < n; i++) {
       if (s.member(l.getAt(i))) {
           t.add(l.getAt(i));
       }
   }
   ```

   The complexity should be expressed in terms of `n`.

   You should express the complexity in the simplest form possible. Apart from the final result you should also describe how you reached it, i.e. show your complexity analysis of the code.

   SOLUTION:

   The loop is repeated `n` times, for `i` from 0 to $\texttt{n} - 1$. For each iteration the following operations are executed in the worst case:

   - 2 calls to `l.getAt`, loopup in dynamic array. Takes $O(1)$.
   - `s.member`, lookup in hash table with perfect hash function. Takes $O(1)$.
   - `t.add`, insertion in AVL tree. For each iteration the tree has maximum size `i`. Takes $O(\log \texttt{i})$.

   The rest inside and outside the loop takes constant time.

   The worst case complexity is

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

2. The following Haskell data type, `Tree`, represents binary unbalanced search trees where the nodes contain integers.

```
data Tree = Empty
          | Node Int Tree Tree

...

removeAllLessThan :: Int -> Tree -> Tree
...
```

An empty (sub) tree is represented by the constructor `Empty`. A node containing the number n is represented by `Node` n l r, where l is the left sub tree and r is the right sub tree.

Implement the function `removeAllLessThan`. The method should, given an integer x and a tree t return a tree which is t but with all nodes removed that contain a number smaller than x. You can assume that the nodes are ordered since it is a search tree and the resulting tree should also be a search tree. The tree is unbalanced, so you do not need to preserve any balance of the tree.

The solution should be given in Haskell, not pseudo code. You may use helper functions but you cannot call any functions that are not implemented in the solution.

SOLUTION:

```
removeAllLessThan x Empty = Empty
removeAllLessThan x (Node y l r) =
 if y >= x then
  Node y (removeAllLessThan x l) r
 else
  removeAllLessThan x r
```

3. Implement a data structure that represents a set of integers. The data structure should have the following operations:

   `empty()` which creates an empty set.

   `add(`x`)` which adds the number x to the set. If x is already in the set it remains unchanged.

   `remove(`x`)` which removes the number x from the set. If x is not in the sets it remains unchanged.

   `contains(`x`)` which returns `true` if the set contains the number x and `false` otherwise.

   `successorOf(`x`)` which returns the smallest integer which is a member of the set and which is larger than x. If there is no such integer then the operation should return `null`. Note that as long as there is a number larger than x in the set then a number should be returned, even if x itself is not member of the set.

   Let n denote the number of integers in the set. The time complexity of the different operations should belong to the following classes:

   - For a **G**: `empty` : $O(1)$, `add` and `remove`: $O(n)$, `contains`: $O(\log n)$, `successorOf`: $O(\log n)$
   - For a **VG**: `empty` : $O(1)$, `add` and `remove`: $O(\log n)$ amortised, `contains`: $O(1)$, `successorOf`: $O(\log n)$

   In both cases you should show your complexity analysis.

   You may either use pseudocode or Java code. If you use pseudocode then make sure to follow the general guidelines at the beginning of the exam.

   You may use the standard data structures and algorithms covered in the course without explaining how they work.

SOLUTION:

```
empty() = let hs be an empty set implemented with a hash table (O(1))
          let ts be an empty set implement with a balanced search tree (O(1))

add(x) = add x to hs (O(1)) amortised
         add x to ts (O(log n))

remove(x) = remove x from hs (O(1)) amortised
            remove x from ts (O(log n))

contains(x) = lookup x in hs (O(1))
```

`successorOf` can be implemented similar to ordinary search in a search tree. Assume that the nodes in the tree are represented by the class `Node` and that there is an instance variable for the root node.

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

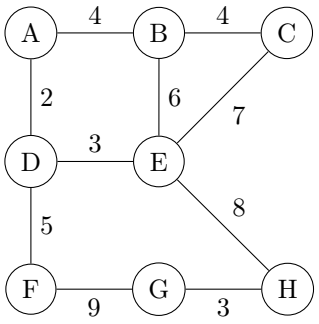
The node class only contains what is necessary to describe the implementation of `successorOf`. The method can be implemented as follows:

```
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);
  }
}
```

The method visits every node in one path from the root. Visiting a node takes $O(1)$. The number of nodes in any path is bounded by $O(\log n)$ since the tree is balanced.

4. Compute a minimal spanning tree for the following graph by manually performing Prim's algorithm using A as starting node.



Your answer should be the set of edges which are members of the spanning tree you have computed. The edges should be listed in the order they are added as Prim's algorithm is executed. Refer to each edge by the labels of the two nodes that it connects, e.g. DF for the edge between nodes D and F.

SOLUTION:

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

5. Construct a data structure that represents a binary relation between integers.

   The data structure should have the following operations:

   `empty()` which creates an object corresponding to an empty relation, i.e. a relation where no pairs of integers are related.

   `addRelation(`x`, `y`)` which makes x become related to y in the relation. If x was previously related to y then nothing happens.

   `removeRelation(`x`, `y`)` which changes the relation so that x is not related to y. If x was initially related to y then nothing happens.

   `isRelated(`x`, `y`)` which returns `true` if x is related to y and `false` otherwise.

   The code below exemplifies how the data structure should work. The values that the boolean variables `b1`, `b2`, `b3`, `b4` and `b5` assume are given as comments at the end of each line.

```
r = empty();
r.addRelation(2, 5);
r.addRelation(4, 3);
r.removeRelation(2, 5);
b1 = isRelated(2, 5);  // false
b2 = isRelated(4, 3);  // true
b3 = isRelated(3, 4);  // false
b4 = isRelated(4, 2);  // false
b5 = isRelated(7, 7);  // false
```

   Note that if x is related to y this does not mean the y is necessarily related to x.

   Let n be the number of integer pairs that are related in the relation. The space complexity for the data structure should belong to $O(n)$, i.e. you may not use more than a linear amount of memory to store the related pairs. The time complexity for all operations should belong to the following complexity class:

   - For a **G**: $O(n)$
   - For a **VG**: $O(1)$ amortised

   In both cases you should show your complexity analysis.

   You may either use pseudocode or Java code. If you use pseudo code then make sure to follow the general guidelines at the beginning of the exam.

   You may use the standard data structures and algorithms covered in the course without explaining how they work.

SOLUTION:

```
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) amortised
    }
    r.get(t).add(u);  // O(1) + O(1) amortised
  } // = 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) amortised
    }
  } // = O(1) amortised

  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)
}
```

6. The following Java class represents a directed graph where the nodes are identified by integers:

```
public class Graph {
    private Map<Integer, List<Integer>> alist;

    public Graph() { ... }
    public void addNode(int x) { ... }
    public void addEdge(int x, int y) { ... }
    public boolean equals(Graph g) { ... }
}
```

The graph is represented by adjacency lists in the instance variable `alist`. The class does not represent multi graphs, i.e. for each ordered pair of nodes, (x, y), there is at most one edge from x to y. Observe that the edge from x to y is not the same as the edge from y to x.

Your task is to implement the constructor and the three operations whose signatures are given above.

`Graph` should create an object representing the empty graph.

`addNode(`x`)` should add a node identified by the integer x. You can assume that a node identified by x does not already exist in the graph.

`addEdge(`x`, `y`)` should add an edge from the node identified by the integer x to the node identified by the integer y. You can assume that the nodes x and y exist in the graph and that there is not already an edge from x to y.

`equals(`g`)` should return `true` if the object g represents the same graph as the current object (`this`). Otherwise it should return `false`. The graphs should not be altered. Two graphs are considered equal if they contain the same set of nodes and edges. Two nodes in two different graphs are considered equal if the have the same integer identifier. Two edges in two different graphs are considered equal if the from-nodes and the to-nodes are pairwise equal.

No other instance variables than the given one, `alist`, may be used.

Let V be the number of nodes and E the number of edges in the graph represented by the current object. The time complexity of the operations should belong to the following classes:

- For a **G**: `addNode`: $O(\log V)$ *not* amortised, `addEdge`: $O(\log V + E)$ *not* amortised, `equals`: $O(V + E)$.
- For a **VG**: `addNode` and `addEdge`: $O(\log V)$ *not* amortised, `equals`: $O(V + E)$.

In both cases you should show your complexity analysis.

You may either use pseudocode or Java code. If you use pseudo code then make sure to follow the general guidelines at the beginning of the exam.

With the exception of graph algorithms, you may use the standard data structures and algorithms covered in the course without explaining how they work.

SOLUTION:

Let's assume that iterators for balanced search trees traverses the tree in-order, which means that if two trees contain the same keys then they will be visited in the same order.

```
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() for the for-loop: O(log V), in total O(V)
      Entry<Integer,List<Integer>> gkv = gait.next();  // O(log V), in total 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) {  // E times in total, max
          // implicit next() for the for-loop: O(1)
        gas.add(y);  // O(1) amortised, O(E) in total
      }
      for (int y : a) {  // E times in total, max
          // implicit next() fot the for-loop: O(1)
        if (!gas.contains(y)) return false; // O(1)
      }
    }
    return true;
  }
}
```

Adding the complexities of each step gives:

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