Seksjon 1

1 OPPGAVE

1 (25 %)

Consider the following fragment of a simple implementation of Union-Find.

```
import java.util.LinkedList; import java.util.ArrayList; import edu.princeton.cs.algs4.StdOut;

public class MyUF {

private int[] id; // id[p] is the identifier of p
public MyUF(int N){ id = new int[N]; for (int i=0; i<N; i++) id[i]=i; }
public int find(int p) { return id[p]; };
public void union(int p, int q) {
  int idp = find(p);
  int idq = find(q);
  if (idp != idq)
    for (int i=0; i<id.length; i++) if (id[i]==idp) id[i]=idq;
}

public boolean isComponentId(int p) { ... } // returns true iff p is identifier of a component

public LinkedList<Integer> componentIdList(){ ... }
// returns all component identfiers in a linked list

public void showComponents(){ ... } // prints for each component all its elements

} // end of class MyUF
```

a) Write the method *isComponentId(int p)* that returns *true* iff *p* is an identifier of a component.

b) Write the method *componentIdList()* that returns all component identfiers in a linked list.

c) Write the method *showComponents()* that prints for each component all the elements of that component.

d) Analyse the run-time of your method *showComponents()*.

Fill in your answer here

2 OPPGAVE

2 (25%)

Consider the following code-fragment:

```
public class MySort{
public static void insertionSort(int[] a){
  for(int i=1; i<a.length; ++i){
    ...
  }
}
```

a) Complete the implementation of *insertionSort*.

b) Give examples where *insertionSort* runs in time linear in the length of *a*.

c) Give examples where *insertionSort* runs in time quadratic in the length of *a*.

d) Explain what ShellSort is and how it improves the run-time performance of *insertionSort*.

Fill in your answer here

3 (25 %)

Consider the following fragment of a simple implementation of TwoThreeTree.

```
public class TwoThreeTree<Key extends Comparable<Key>> {
private Node root;
private class Node{
  private Key key1;
  private Key key2;
  private Node left, mid, right;
  public Node(Key k1, Key k2, Node l, Node m, Node r){
    key1 = k1; key2 = k2;
    assert key1!=null && (key2==null || key1.compareTo(key2) < 0);
    left = l; mid = m; right = r;
    assert key2!= null || mid==null ;}
} // End of class Node

public TwoThreeTree(Key k1, Key k2, Node l, Node m, Node r) {
  root = new Node(k1,k2,l,m,r); }

public Node search(Key k, Node r){ ... }
// returns a Node under r in which k occurs, if such a Node exists; returns null otherwise

} // End of class TwoThreeTree
```

An object of this class represents a so-called 2-3-tree, satisfying the data invariants as expressed in the assertions of the Node-constructor. A 2-Node is a *Node* with one key and two children; a 3-Node is a *Node* with two keys and three children. Children can be *null*.

a) Which of the following give correct *TwoThreeTree*-objects: *TwoThreeTree(0,null,null,null,null); TwoThreeTree(1,0,null,null,null); TwoThreeTree(null,0,null,null,null); TwoThreeTree(-1,0,null,null,null); TwoThreeTree(0,null,0,null,0)*?

A 2-3-tree is a 2-3 *search* tree if the following additional invariant holds: in every 2-Node, all keys in *left* are smaller than *key1*, and all keys in *right* are greater than *key1*; in every 3-Node, all keys in *left* are smaller than *key1*, and all keys in *mid* are between *key1* and *key2*, and all keys in *right* are greater than *key2*.

b) Assuming *Node r* is a 2-3 search tree, write a method *search(Key k, Node r)* that returns the Node under *r* in which *k* occurs if such a Node exists, and returns *null* otherwise.

c) Let *N* be the number of keys in a 2-3 search tree. Give the worst-case run-time of your answer under b).

d) How could the worst-case run-time under c) be improved?

Fill in your answer here

4 OPPGAVE

4 (25%)

Given a directed graph *G* with nodes *0,...,V-1,* represented by adjacency lists of out-edges. More precisely, for each node *v*, *adj[v]* is a linked list which contains all nodes *w* such that *G* has an arrow from *v* to *w*.

a) Write a method *boolean reachable(int s, int t)* that returns a boolean *true* if and only if *G* contains a path from source *s* to target *t*.

b) (Bonus: +10%) Write a method *boolean acyclic()* that returns a boolean *true* if and only if *G* is acyclic.

Fill in your answer here