DIT960 – Datastrukturer

suggested solutions for exam 2017-06-03

1. The following code takes as input two arrays with elements of the same type. The arrays are called `a` and `b`. The code returns a dynamic array which contains one copy of all elements which are in both `a` and `b`.

```
s = new empty set implemented as an AA tree
c = new empty dynamic array

for every element x in a
  s.insert(x)

for every element x in b
  if s.member(y) then
    c.append(y)
    s.remove(y)

return c
```

What is the big-O time complexity of this code? Express it in terms of n, the maximum of the lengths of the arrays. 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:

There is a typo in the question. The head of the second loop should read:

```
for every element y in b
```

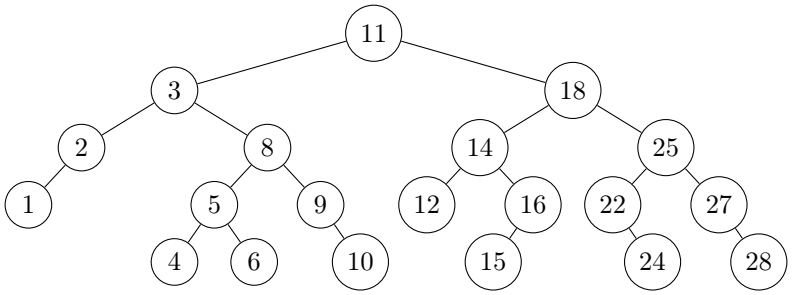
Initialisation of `s` and `c`: $O(1)$.

First loop is repeated at most n times. Every iteration is an insertion into a AA tree of size at most n which takes $O(\log n)$. So, first loop: $O(n \log n)$.

Second loop is repeated at most n times. Every iteration is, in worst case, a lookup in an AA tree of size at most n ($O(\log n)$), an append to a dynamic array ($O(1)$ amortised) and a remove in an AA tree of size at most n ($O(\log n)$). In total for second loop: $O(n(\log n + 1 + \log n)) = O(n \log n)$ (not amortised because there the dynamic array is empty to begin with so a sequence of at most n appends takes $O(n)$).

In total: $O(1 + n \log n + n \log n) = O(n \log n)$.

2. Let T be the following binary search tree (BST):



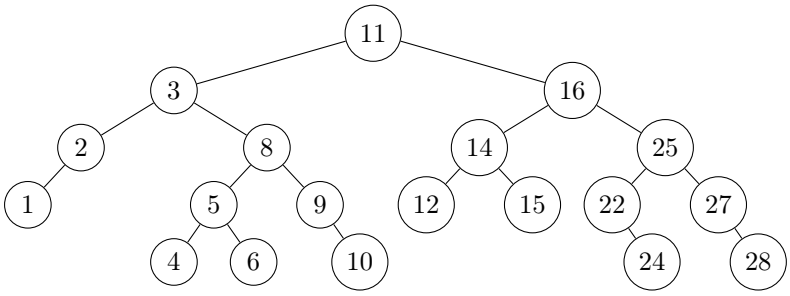
The order is the normal one for integers. For a **G** you need to solve the first sub question below. For a **VG** you need to solve both sub questions.

(a) Assume that T is an unbalanced BST. How does the tree look after deleting the number 18? You only need to show the resulting tree, not the intermediate steps.

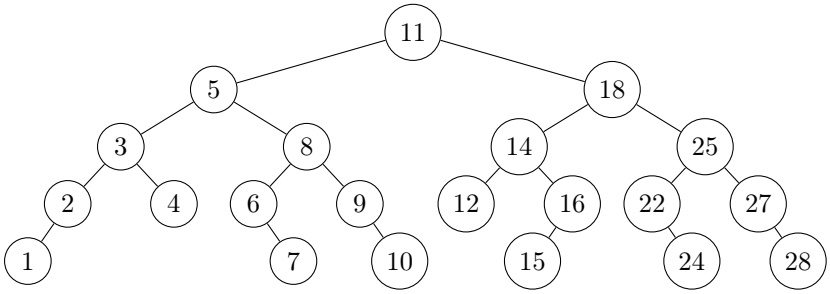
(b) Assume that T is an AVL tree. How does the tree look after inserting the number 7? You only need to show the resulting tree. Notice that you should start with T, not the resulting tree in the previous question.

SOLUTION:

(a) 18 has two sub trees so it must be replaced by either its predecessor or successor. Replacing it with its predecessor results in the tree:



(b) After a standard BST insertion (7 becomes right child of 6) the node with 3 becomes unbalanced. Performing a right-left rotation results in the balanced tree:

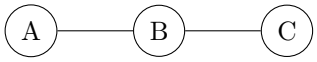


3. The following Java type is used to represent an undirected graph with string labels on the nodes/vertices and no labels on the edges:

```
HashMap<String, ArrayList<String>>
```

It is a adjecency list representation. Remember that for undirected graphs there are two references for each edge between a node i and a node j; one from from i to j and one from j to i. This is part of the invariant of the representation.

As an example, the graph



is e.g. represented by the object $\{\mathrm{A} \to [\mathrm{B}], \mathrm{B} \to [\mathrm{A}, \mathrm{C}], \mathrm{C} \to [\mathrm{B}]\}$.

Implement the operation

```
boolean isTree(HashMap<String, ArrayList<String>> g)
```

which determines whether the graph `g` is a tree, i.e. is connected and acyclic. You can assume that `g` satisfies the invariant.

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 covered in the course without explaining how they work. You may also use sorting algorithms without explanation, but the implementation of graph algorithms must be showed explicitly.

For a **VG** you should also analyse the time complexity of your code. You may state the complexity of standard operations covered in the course (except graph algorithms) without motivation.

SOLUTION:

```
isTree(g)
  if g.size() == 0 then return true
  let vis = empty hash set of strings
  let n = arbitrary node, e.g. first object returned by iterator of keys in g
  if dfs(g, null, n) then return false
  return vis.size() == g.size()  // if size is the same that g is connected

// dfs performs a depth first search of the graph and
// stops and returns true if a cycle is found
// to know if a cycle is found it keeps track of the previous node
dfs(g, prev, cur)
  vis.add(cur)
  for each string succ in g.get(cur) // for adjecent node of cur
    if vis.contains(succ) then
      if not succ.equals(prev) then  // if it's visited and not the node we just came from
        return true  // a cycle is found
    else
      if dfs(g, cur, succ) then return true
  return false
```

Let V be the number of nodes and E the numbers of edges.

`isTree` takes $O(1)$ apart from the call to `dfs`.

`dfs` is called at most once for each nodes. Apart from the loop each execution of `dfs` takes $O(1)$ (insertion and lookup in hash tables). The loop is repeated, in total for all executions of `dfs`, at most twice for each edge. Each iteration takes, apart from the recursive call, $O(1)$ (lookup in hash table, lookup in array)

In total: $O(1 + V \cdot 1 + 2E \cdot 1) = O(V + E)$

Alternative solution: Perform a traversal which only looks for connectedness. Then count the number of edges. The sum of the sizes of all lists adjecent nodes should be $2(V - 1)$.

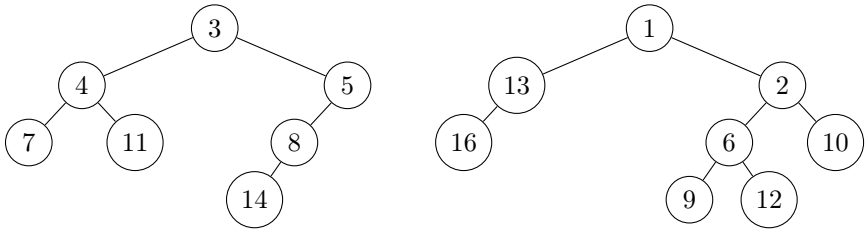
4. For a **G** you need to solve either one of the sub questions below. For a **VG** you need to solve both sub questions. In both sub questions the order of the elements is the normal one for integers.

   (a) The following array represents a binary heap in the standard way:

5	7	9	11	8	12	15	17	15	20	22	16	17	19	17	18	20	17	18

   What does the array look like after performing a delete-min on it?

   (b) The following two trees represent skew heaps:



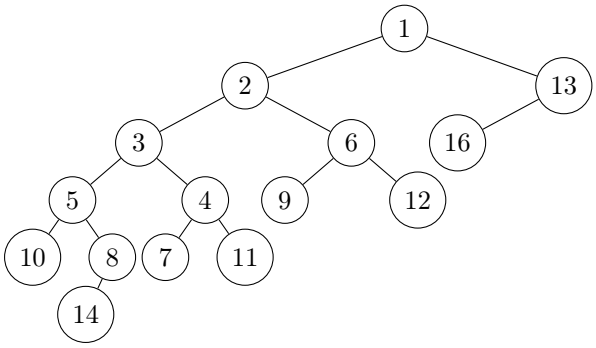
   How does the resulting tree look after (skew heap) merging the two trees?

SOLUTION:

(a) It's probably easiest to get it right by drawing the heap as a tree, performing delete-min and then translating back to an array.

7	8	9	11	18	12	15	17	15	20	22	16	17	19	17	18	20	17

(b) The merged tree is:



5. Implement an operation that takes a non-empty array of elements with defined equality and returns a value which occurs at least as many times as any other value. As an example, for the array $\{5, 2, 3, 8, 2, 1, 3, 12, 3, 2\}$ the answer would be 2 or 3. The implementation should be generic. It should not be specialised to arrays of integers.

   For a **G** the time complexity of the operation should be $O(n \log n)$, where n is the size of the array.

   For a **VG** the time complexity should be $O(n)$ provided that the appropriate conditions for the element type are satisfied. For a **VG** you should also motivate why the complexity requirement is satisfied.

   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:

```
assume arr is the input array

let count = new empty hash map from integers to integers

for each integer x in arr
  if count.contains(x) then
    count.put(x, count.get(x) + 1)
  else
    count.put(x, 1)

int maxcount = 0
int mostfrequentvalue = any number  // this value will be overwritten since arr is nonempty
for each integer x in count.keySet()
  if count.get(x) > maxcount then
    maxcount = count.get(x)
    mostfrequentvalue = x

return mostfrequentvalue
```

   Initialisation of `count` takes $O(1)$.

   First loop is repeated n times and each iteration takes $O(1)$ (hash table operations).

   Second loop is repeated at most $O(n)$ times since the number of entries in `count` is at most n and the size of the hash table is at most $c \cdot n$ where c is a constant determined by the hash table growth factor and maximal load factor. Each iteration takes $O(1)$ (hash table operations).

   In total: $O(1 + n \cdot 1 + n \cdot 1) = O(n)$

   A solution that takes $O(n \log n)$ is to first sort the array and then loop over each element of it and find a number with the most consecutive occurrences.

6. Implement a data structure for a collection of elements which has the following constructor and operations:

   `empty` which creates a collection containing no elements

   `add(x)` which inserts element `x` into the collection

   `deleteOldestMin()` which returns the minimum element and deletes it from the collection. If there are several minimum elements then the operation should return and delete the element, among the minimum ones, that was first added to the collection.

   The implementation should be generic, but you can of course assume that the elements can be pairwise compared for less-than, equal, greater-then.

   For a **G** the time complexity of `add` and `deleteOldestMin` should be $O(n)$ (or better), where n is the number of elements in the collection.

   For a **VG** the time complexity of `add` and `deleteOldestMin` should be $O(\log n)$.

   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:

Let the generic element type be `T`. Let the representation be

```
An integer k.
A priority queue q implemented by a binary heap.
  The queue contains pairs of type <T, int>.
  The order (comparator) use in the queue first compares the first element (of type \verb!T!).
  If the elements are unequal than the order for the pairs is the order of the first elements.
  If the elements are equal than the order for the pairs is the order of the second elements (the integers).
  Smaller integer means higher priority (as usual).
```

Implement the constructor and operations like this.

```
empty:
  let k = 0
  let q = empty binary heap

add(x):
  q.add(<x, k>)
  k = k + 1

deleteOldestMin():
  let p = q.deleteMin()
  return first component of p (the value of type T)
```

By defining the order of elements in the priority queue like that and increasing `k` for each insertion the oldest element will be deleted when there are several equal elements (because it will be paired with the lowest integer among them).

This solution has $O(\log n)$ for both `add` and `deleteOldestMin`.

A solution that has $O(1)$ for `add` and $O(n)$ for `deleteOldestMin` is to have a list of elements, let `add` append each new element to the list and let `deleteOldestMin` loop through all elements in the list and find the first smallest one.