Exam (with answers)
Data structures DIT960

Time	Monday 30th May 2016, 14:00–18:00
Place	Hörsalsvägen
Course responsible	Nick Smallbone, tel. 0707 183062

The exam consists of **six questions**. For each question you can get a **G** or a **VG**.

To get a **G** on the exam, you need to answer **three** questions to **G** standard.

To get a **VG** on the exam, you need to answer **five** questions to **VG** standard.

A fully correct answer for a question will get a VG. An answer with small mistakes will get a G. An answer with large mistakes will get a U.

When a question asks for **pseudocode**, you can use a mixture of English and programming notation to describe your solution, and should give enough detail that a competent programmer could easily implement your solution.

Allowed aids	One A4 piece of paper of notes, which should be handed in after the exam. You may use both sides. You may also bring a dictionary.
Note	Begin each question on a new page. Write your anonymous code (*not* your name) on every page.

Good luck!

1. The following algorithm takes as input an array, and returns the array with all the duplicate elements removed. For example, if the input array is {1, 3, 3, 2, 4, 2}, the algorithm returns {1, 3, 2, 4}.

```
S = new empty set
A = new empty dynamic array
for every element x in input array
  if not S.member(x) then
    S.insert(x)
    A.append(x)

return A
```

What is the big-O complexity of this algorithm, if the set is implemented as:

a) an AVL tree?

For G: O(n log n)
(The loop runs n times, and member + insert takes O(log n) time. Append takes amortised O(1) time so the sequence of n appends takes O(n) time.)

For VG: O(n log m) – the set S always contains at most m elements

b) a hash table?

Answer: O(n) because hash table operations take (expected) O(1) time

For G: write the complexity in terms of *n*, the size of the input array.

For VG: write the complexity in terms of *n* and *m*, where *n* is the size of the input array and *m* is the number of distinct elements in the array (i.e. the number of elements ignoring duplicates).

2. Suppose you have the following hash table, implemented using *linear probing*. The hash function we are using is the identity function, $h(x) = x$.

0	1	2	3	4	5	6	7	8
9	18		12	3	14	4	21	

a) In which order could the elements have been added to the hash table? There are several correct answers, and you should give all of them. Assume that the hash table has never been resized, and no elements have been deleted yet.

A	9, 14, 4, 18, 12, 3, 21
B	12, 3, 14, 18, 4, 9, 21
C	12, 14, 3, 9, 4, 18, 21
D	9, 12, 14, 3, 4, 21, 18
E	12, 9, 18, 3, 14, 21, 4

Answer: C and D

In A, 4 would be inserted at index 4 instead of 6
In B, 18 would be inserted at index 0 instead of 1
In E, 21 would be inserted at index 6 instead of 7

b) Remove 3 from the hash table, and write down how it looks afterwards.

Answer: the important thing is to use lazy deletion (just removing 3 from the array will leave e.g. 4 in the wrong cluster). So we get this:

0	1	2	3	4	5	6	7	8
9	18		12	XXX	14	4	21	

c) **For VG only:** if we want a hash table that stores a set of *strings*, one possible hash function is the string's length, $h(x) = x.length$.

Is this a good hash function? Explain your answer.

Answer: no. Strings with the same length will have the same hash code. If we insert lots of strings with the same length, lookup will take O(n) time instead of O(1).

[This hash function was apparently used by early versions of PHP! See: https://lwn.net/Articles/577494/]

3. Design an algorithm that takes two arrays, and returns *true* if the arrays are disjoint, i.e. have no elements in common.

   You may freely use standard data structures and algorithms from the course in your solution, without explaining how they are implemented.

   Write down your algorithm as **pseudocode**. You don't need to write Java code, but be precise – a competent programmer should be able to take your description and easily implement it.

   For a G: your algorithm should take O(n log n) time.

   For a VG: your algorithm should take O(n log m) time, where n is the size of the *larger* array and m is the size of the *smaller* array.
   Hint for VG: since n ≥ m, this is the same as O(n log m + m log m).

   For a G:
   Suppose the two arrays are called array1 and array2. The idea is to insert all elements of array1 into an (e.g.) AA tree and then check if any element of array2 is in the tree.

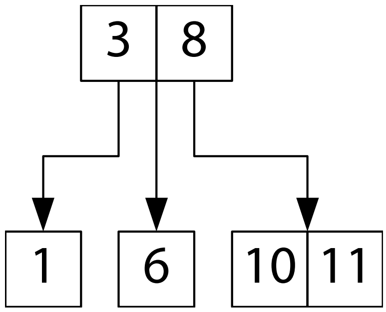
```
S = new AA tree
for x in array1 do
        S.insert(x)
for x in array2 do
        if S.member(x) then return false
return true
```

   For a VG:
   Since S contains the contents of array1, the algorithm above takes O(m log m + n log m) time, where m is the size of array1 and n is the size of array2. To get the required time complexity, we just need to make sure that array1 is the *smaller* array. So, we do:

```
        if array1.size() > array2.size() then
                swap array1 and array2 variables
```

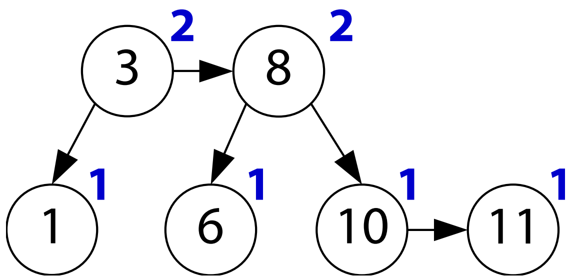
   and then run the previous algorithm.

4. Look at the 2-3 tree below.



a) Draw the 2-3 tree as an AA tree. Mark each node with its level.[1]

Answer (I've drawn nodes of the same level next to each other but you don't need to):

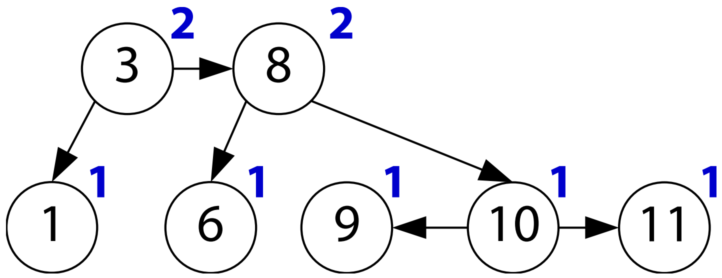


b) Insert 9 into the AA tree using the AA tree insertion algorithm. Write down the final tree.

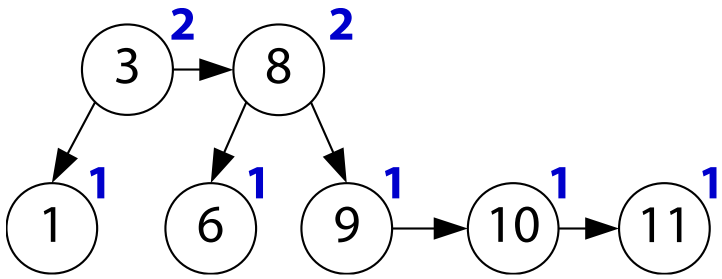
I've written down each step but it's enough to just give the final answer.

1 If you have studied the version of AA trees that uses colours, you may write down the colours instead of the levels.

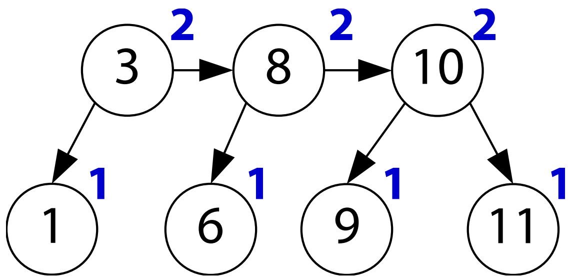
First we do BST insertion, giving the new node a level of 1:



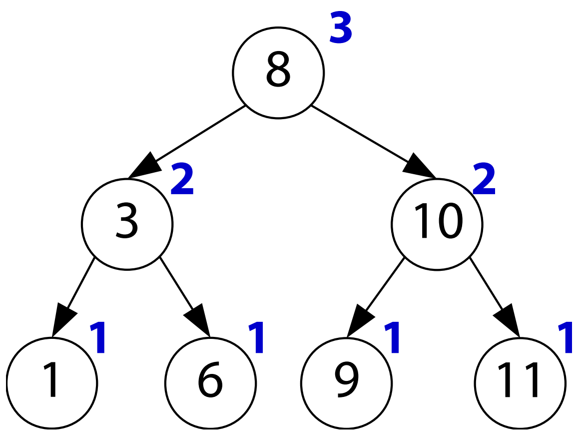
Now 10 has the same height as its left child, so we do a skew (rotation):



Now 10 has become a “4-node”, so we do a split, which lifts 10 up:



Now 3 has become a “4-node”, so we do another split, which lifts 8 up:



5. A *bidirectional map* is a map which supports bidirectional lookup: given a key, you can find the corresponding value, and given a value, you can find the corresponding key.

   In a bidirectional map there is always a one-to-one relationship between keys and values. In other words, each key has exactly one value, and each value is found under exactly one key.

   It supports the usual map operations:

   - `new()`: create a new, empty map
   - `insert(k, v)`: add the mapping $k \to v$ to the map;
     any existing mapping with key k or value v is removed.
   - `lookup(k)`: if the map contains a mapping $k \to v$, return v
   - `delete(k)`: if the map contains a mapping $k \to v$, delete it

   plus the following *reverse lookup* operation:

   - `rlookup(v)`: if the map contains a mapping $k \to v$, return k

   The following example shows what the various operations do.

Operation	Result
`new()`	Map is {}
`insert(1,2)`	Map is $\{1 \to 2\}$
`insert(3,4)`	Map is $\{1 \to 2, 3 \to 4\}$
`lookup(1)`	Returns 2
`insert(4,2)`	Map is $\{3 \to 4, 4 \to 2\}$. Notice that the mapping $1 \to 2$ is replaced by $4 \to 2$.
`rlookup(2)`	Returns 4
`delete(4)`	Map is $\{3 \to 4\}$

We can implement a bidirectional map using *two* maps, each implemented as e.g. an AA tree:

- `forward` is a map from keys to values.
  In the example above, it contains 3 → 4 and 4 → 2.

- `back` is a map from values to keys.
  In the example above, it contains 4 → 3 and 2 → 4.

The invariant is that the two maps always contain the same data: `forward` contains the mapping $k \to v$, if and only if `back` contains the mapping $v \to k$.

We can then implement `new`, `lookup` and `rlookup` as follows:

- `new`: set `forward` and `back` to be empty AA trees

- `lookup(k)`: look up k in `forward` using the BST lookup algorithm

- `rlookup(v)`: look up v in `back` using the BST lookup algorithm

Your task is to implement the remaining operations, `insert` and `delete`, with O(log n) complexity.

Be careful: the algorithm is more complicated than it seems. Be extra careful that you maintain the data structure invariant. It's also a good idea to test your solution on the previous page's example.

Give **pseudocode** for each of the operations. You don't need to write Java code, but be precise – a competent programmer should be able to take your description and easily implement it..

You may freely use standard data structures and algorithms from the course in your solution, including insertion/lookup/deletion in a map, without explaining how they are implemented.

I'll assume that lookup and rlookup return null when the key is not found. The tricky thing is to always keep the forward and backward maps in sync.

```
delete(k):
    // Find the value so that we can delete it from the backward map
    v = forward.lookup(k)
    if v != null then
        forward.delete(k)
        back.delete(v)
```

In insertion, we need to find any existing entries with the same key or value, and remove them from the forward and backward maps.
On the example in the question, when we call insert(4, 2), that should remove 1 from the forward map and 2 from the backward map.

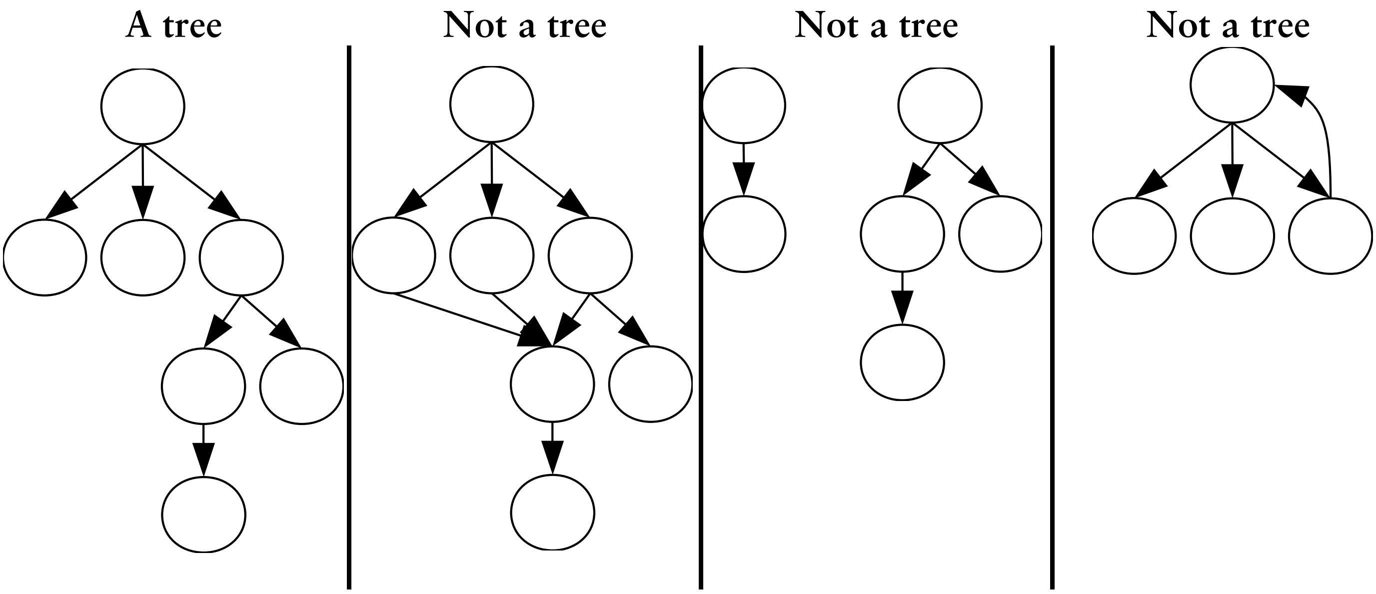
```
insert(k, v):
    // Find and remove any entry with key k
    v' = forward.lookup(k)
    if v' != null then
        forward.delete(k) // this line is not strictly necessary
        back.delete(v')

    // Find and remove any entry with value v
    k' = back.lookup(v)
    if k' != null then
        forward.delete(k')
        back.delete(v) // this line is not strictly necessary

    forward.insert(k, v)
    backward.insert(v, k)
```

6. We can think of a tree as being a special kind of directed graph. To model a tree as a graph, we make the nodes of the tree become nodes in the graph, and draw an edge from a parent node to each of its children.

   The drawing on the left shows a tree as a graph; the other three directed graphs do *not* correspond to a tree.

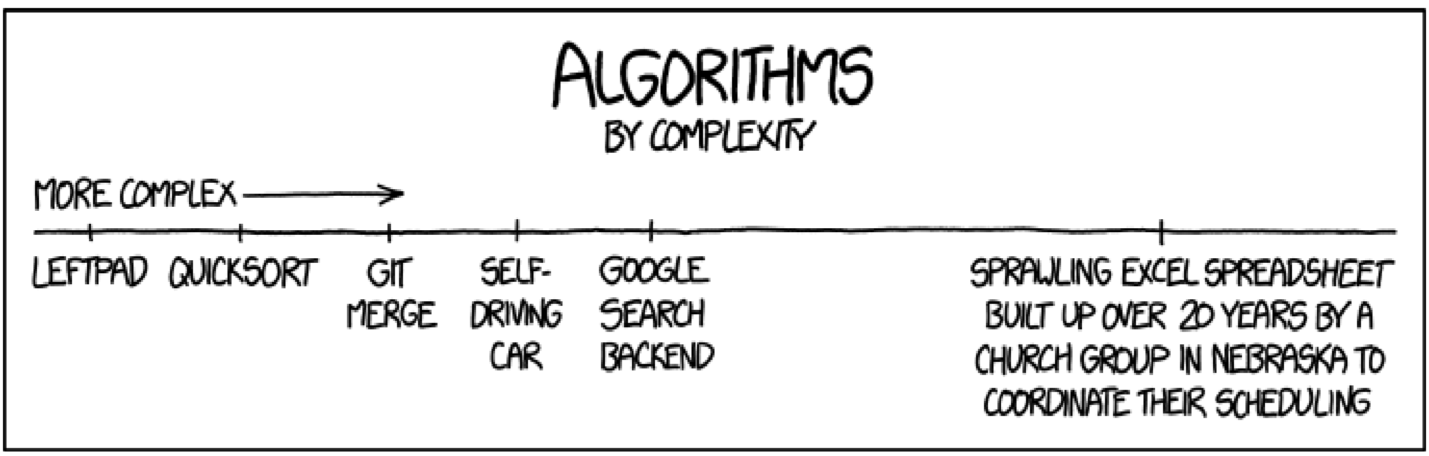


   Suppose we want to check if a given directed graph corresponds to a tree. What properties should we check that the graph has? Write down a list of properties such that, if a directed graph has those properties, it must be a tree. You can refer to standard properties of graphs in your answer without explaining them.

   I answered this question like so:

   - the graph must be (weakly) connected
   - it must be acyclic
   - each node must have 0 or 1 predecessors (there must be at most one edge to each node)

But there are many other ways to answer it. For example, you can say that there must be one node (the root) such that there is exactly one path from the root to any other node.



http://www.xkcd.com/1667