Exam
Data structures DAT036/DAT037/DIT960

Time	Thursday 18th August 2016, 08:30–12:30
Place	Maskinhuset / SB Multisal
Course responsible	Nick Smallbone, tel. 0707 183062

The exam consists of **six questions**. For each question you can get a **3, 4 or 5.** You may answer in English or Swedish.

To get a **3** on the exam, you need to get a **3** on **3** questions.
To get a **4** on the exam, you need to get a **4** on **4** questions.
To get a **5** on the exam, you need to get a **5** on **5** questions.

For GU students, a **G** corresponds to a 3, and a **VG** corresponds to a 5.

A fully correct answer for a question will get full marks. An answer with small mistakes may get a lower grade. An answer with large mistakes will get a U.

When a question asks for **pseudocode**, you can use a mixture of natural language and programming notation in 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 which may contain duplicates. It returns `true` if all elements of the array occur an odd number of times. Otherwise it returns `false`.

   For example, on the array `{1, 3, 2, 2, 5, 2}` it returns `true`, but on the array `{1, 3, 2, 2, 5, 2, 5}` it returns `false` because 5 occurs an even number of times.

```
S = new AVL tree
for every element x in input array
  if S.member(x)
    S.delete(x)
  else
    S.insert(x)
// At this point, S contains those elements that
// occur an odd number of times

for every element x in input array
  if not S.member(x)
    return false
return true
```

   What is the big-O complexity of this algorithm?

   For a 3: express your answer in terms of n, the length of the input list.
   For a 4: express your answer in terms of n and m, where n is the length of the input list and m is the number of *distinct* elements in the input list.

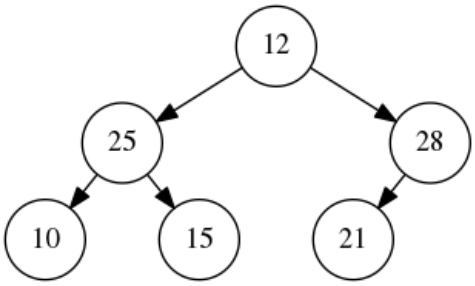
   Answer:

   For a 3: O(n log n). There are O(n) tree operations and each takes O(log n) time.

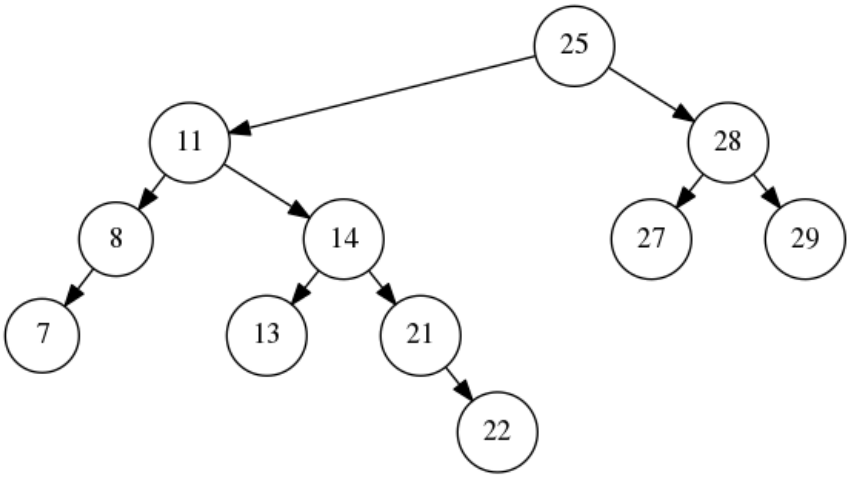
   For a 4: O(n log m). The AVL tree never contains more than m elements. So there are O(n) tree operations and each takes O(log m) time.

2. Have a look at the following three binary trees.

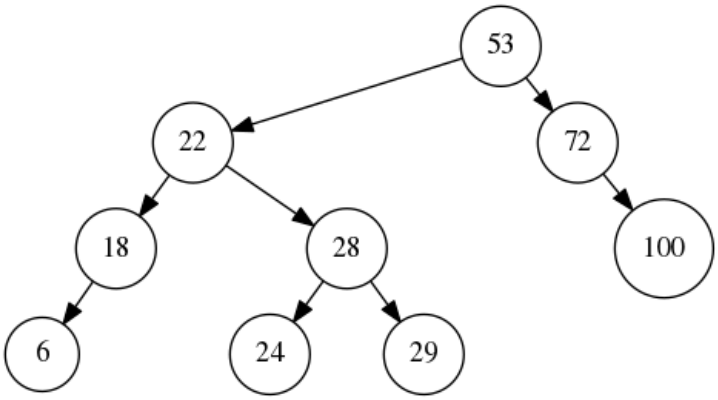
Tree A:



Tree B:



Tree C:



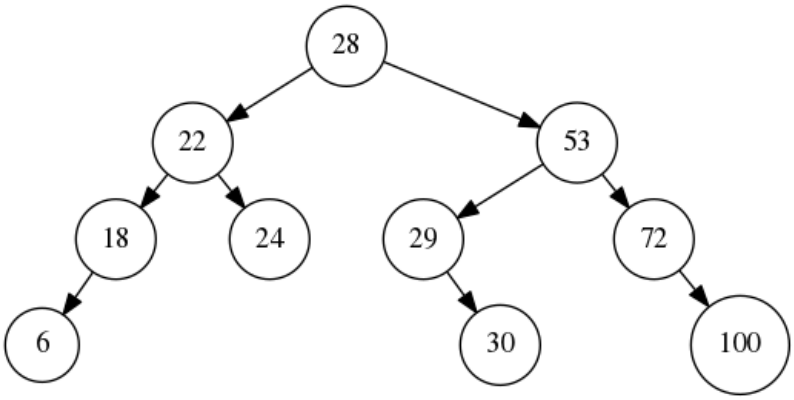
a) One of these trees is an AVL tree. Which one?

b) Insert 30 into the tree using the AVL insertion algorithm. Write down the final tree.

Answer:

a) C is an AVL tree. (A is not a binary search tree as 25 > 15. B is not an AVL tree as the root's left child has a height of 4 but the root's right child has a height of 2.

b) After inserting 30 using BST insertion, the tree is unbalanced (root's left child height = 4, right child height = 2). It's a left-right tree so a double rotation fixes it:



3. Suppose we are implementing a class for dynamic arrays in Java:

```
class DynamicArray<A> {
  private int size;
  private A[] data;

  ... class methods go here ...
}
```

The class contains two fields, seen above: `size` is the number of elements in the dynamic array, and `data` is the contents of the dynamic array, stored in indices `data[0], ..., data[size-1]`. As usual with dynamic arrays, there may be unused space at the end of the array, in which case `size < data.length`.

You should answer the questions below with **Java code, not pseudocode.** You do not need to check for errors. For example, you can assume that the array is not empty and the index is a valid index in the array. You do not need to resize the array if it becomes too small.

a) Write a method `void deleteLast()` which removes the last element from the dynamic array. It should take **O(1)** time.

```
void deleteLast() {
   size--;
}
```

b) Write a method `void delete(int index)` which removes the element at position `index` from the dynamic array, preserving the order of elements in the array. For example, calling `delete(2)` on a dynamic array containing `{2,5,4,1,3}` should result in `{2,5,1,3}`.

Your method should take **O(n)** time.

Only completely correct solutions will be accepted. You may want to test your code on the example above to make sure it works.

```
void delete(int index) {
    size--;
    for (int i = index; i < size; i++)
    data[i] = data[i+1];
}
```

The idea is to move all elements after index down by 1.

c) **For a 4:** If we do not need to preserve the order of elements in the array, we can implement deletion more efficiently. Write a method `void deleteUnordered(int index)` which removes the element at position `index` from the dynamic array. It is allowed to alter the order of elements in the array.

Your method should take **O(1)** time.

```
void deleteUnordered(int index) {
    data[index] = data[size-1];
    size--;
}
```

The idea is to overwrite the deleted element with the last element in the array.

4. Design a data structure for storing a set of integers. It should support the following operations:
   - `new()`: create a new, empty set
   - `insert(x)`: add an integer `x` to the set
   - `member(x)`: test if a given integer `x` is in the set
   - `increaseBy(x)`: add `x` to all the integers in the set

   For example, after calling `increaseBy(2)` on a set containing 1,2,3,4,5, the set afterwards should contain 3,4,5,6,7.

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

   You should say what design or existing data structure you have chosen, and write down the operations 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.

   The operations must have the following time complexities. **You must also briefly justify why each operation has the right complexity.** If you use a standard algorithm, you can assume its complexity without justification.

   - **For a 3**:
     O(1) for `new`,
     O(log n) for `insert/member`,
     O(n) for `increaseBy`
     (where n is the number of elements in the set)
   - **For a 5:**
     as for G but the complexity of `increaseBy` must be O(1).

Answer:

We can use an AVL tree.

- new(): create a new AVL tree
- insert(x): use AVL insertion
- member(x): use BST membership testing
- increaseBy(x): use (e.g.) an inorder traversal to iterate through all nodes of the tree. For each node, add x to the value. Note that this preserves the relative order of all elements so will note break the AVL invariant.

For a 5:

We maintain both an AVL tree and an integer k which stores the total of all calls to increaseBy.

- new: create a new AVL tree, set k=0
- insert(n): insert n-k into AVL tree
- member(n): check if n-k is contained in AVL tree
- increaseBy(n): set k=k+n

5. Suppose we are given the following Haskell type of binary trees containing integers:

```
data Tree = Nil | Node Integer Tree Tree
```

Write a function `isBST :: Tree → Bool` which takes a binary tree and returns `True` if it is a binary search tree.

For a 3: your function should have $\mathbf{O(n^2)}$ complexity, where n is the size of the tree. **Hint:** you will probably need to define some helper functions.

For a 5: your function should have $\mathbf{O(n)}$ complexity. **Hint:** try defining a function which returns whether the tree is a BST together with some other information.

Answer:

Here is one solution, which defines a helper function `values :: Tree → [Integer]` returning all values in a tree.

```
isBST Nil = True
isBST (Node x l r) = all (<= x) (values l) && all (>= x)
(values r)

values Nil = []
values (Node x l r) = values l ++ [x] ++ values r
```

For a 5:

We define a helper function

```
isBST' :: Maybe Integer → Maybe Integer → Tree → Bool
```

which takes a tree and a lower and upper bound, and checks that the tree is a BST and all of its values are within the bounds.

```
isBST' _ _ Nil = Nil
isBST' min max (Node x l r) =
```

```
    checkMin min x && checkMax max x &&
    isBST' min (Just x) l && isBST' (Just x) max r
    where
      checkMin Nothing _ = True
      checkMin (Just min) x = min <= x
      checkMax Nothing _ = True
      checkMax (Just max) x = x <= max
```

Using this funciton we can define `isBST`:

```
isBST t = isBST' Nothing Nothing t
```

6. A *double-ended priority queue* is a priority queue that supports removing both the minimum and the maximum element. It provides the following operations:

   - `insert` – add an element to the priority queue
   - `findMin/deleteMin` – find or delete the minimum element
   - `findMax/deleteMax` – find or delete the maximum element

   While writing a program, you discover you need a double-ended priority queue. Your friend suggests a way to implement one:

   > Maintain two priority queues, one of them a min heap and the other a max heap[1]. To insert an item, insert it into both heaps. To implement `findMin/deleteMin` simply call `findMin/deleteMin` on the min heap. To implement `findMax/deleteMax` call `findMax/deleteMax` on the max heap.

   The following Java code illustrates the idea:

```
MinHeap minheap = new MinHeap();
MaxHeap maxheap = new MaxHeap();
void insert(E x) { minheap.insert(x); maxheap.insert(x); }
E findMin() { return minheap.findMin(); }
E findMax() { return maxheap.findMax(); }
void deleteMin() { minheap.deleteMin(); }
void deleteMax() { maxheap.deleteMax(); }
```

   Unfortunately, this idea does not work. Once you see why, write down a sequence of operations where this design would give the wrong answer.

   Answer:

   The problem is that in deleteMin, you need to remove the minimum element from both minheap and maxheap (likewise for deleteMax). The following example goes wrong:

```
insert(1); // minheap contains 1, maxheap contains 1
deleteMax(); // minheap contains 1, maxheap is empty
insert(2); // minheap contains 1 and 2, maxheap contains 2
findMin(); // returns 1, which should not be in the priority queue!
```

---

1 Recall that a max heap supports the operations `insert`, `findMax` and `deleteMax`. A min heap is an ordinary binary heap and supports `insert`, `findMin` and `deleteMin`.

For a 4:

A *min-max heap* is a binary tree with the following invariant:

- The value of any node at an *even* level in the tree is *less than or equal to* all values in the node's subtree;
- the value of any node at an *odd* level in the tree is *greater than or equal to* all values in the node's subtree.

The *level* of a node is defined as follows: the root of the tree is at level 0, its children are at level 1, its grandchildren are at level 2, and so on.

Describe how to find the minimum and maximum elements in a min-max heap. Make sure you consider the case where the heap has one element. You do not have to worry about insertion or deletion, only `findMin/findMax`.

Answer:

Drawing a few min-max heaps you will see that the least element must be the root (like in a min heap), and the greatest element must be one of the root's children. There is one exception: if the heap only contains one element, then the root is both the least and the greatest element.

So to find the minimum element: just return the root's value.

To find the maximum element, look at the root, the root's left child and the root's right child, and return whichever of the three is biggest.

(More information: http://en.wikipedia.org/wiki/Min-max_heap)