Exercises 4

Please, include a comment with your full name and degree at the beginning of any file you upload with your solutions.

1. (Java) Implement priority queues using a sorted (in ascending order) linked list of nodes, just like the Haskell implementation in the slides.

2. (Haskell) Implement priority queues using a lineal recursive data type so that elements are kept unsorted. Your implementation should place last inserted element at the front (just like a stack). `first` and `dequeue` should locate minimum element in unsorted structure and, respectively, remove or return it. Analyse performance of operations for this implementation.

3. (Java) Solve previous exercise in Java using an unsorted linked list.

4. (Java) Write a program to determine average height of binary search trees with n random elements. Use `java.util.Random` to generate n elements to insert into an initially empty tree. Repeat this procedure to build 50 random trees (all of them with n elements) and calculate their average height. Using this method, determine average height for binary search trees with 10, 100 and 1000 random elements.

5. (Java) Solve previous exercise, this time to determine average height of random AVL trees.

6. (Haskell). Library `DataStructures.Util.Random` (available with source code of examples for unit 4) provides function `randomsR :: (Random a) => (a,a) -> Seed -> [a]` that can be used to generate an infinite list of random values within a range. The second parameter is an `Int` value (`Seed` is a type synonym of `Int`) so that, for different values of the seed, the function returns different random sequences. For instance, we can simulate rolling 5 times a dice with the following function:

```
fiveRolls :: Seed -> [Int]
fiveRolls seed = take 5 (randomsR (1,6) seed)
```

And use different seeds to do different simulations for that experiment:

```
Main> fiveRolls 0
[3,5,3,1,1]
Main> fiveRolls 1
[4,3,4,1,5]
Main> fiveRolls 2
[6,1,1,4,1]
Main> [ fiveRolls s | s <- [0..2]]
[[3,5,3,1,1],[4,3,4,1,5],[6,1,1,4,1]]
```

Solve problems 4 and 5 in Haskell using `DataStructures.Util.Random` library. To generate 50 different random trees (use seed values in [0..49] and a list comprenhesion).

7. (Haskell) Write a function:

```
levels :: Tree a -> [a]
```

to traverse elements in a tree by levels. First element in resulting list should correspond to element at level 0 (root), followed by elements at level 1, followed by elements at level 2, etc. To implement this algorithm, you will need to use a queue of trees that maintains yet to be traversed sub-trees. Start by visiting root node, and adding all its children to queue. Continue by extracting first node from queue and repeating previous procedure until queue gets empty.

8. (Haskell) This exercise was taken from http://programmingpraxis.com.
Maxiphobic heaps are a variant of leftist heaps. Like leftist heaps, maxiphobic heaps are represented as binary trees arranged according to the heap property that every element is less than or equal to its two children. `minElem` looks at the root of the tree, `delMin` discards the root and merges its two children, and `insert` merges the existing tree with a singleton tree containing the new element.
The key to leftist trees and maxiphobic trees is the merge operation. In leftist trees, the weight of each left child is always greater than or equal to the weight of its sibling, and the merge operation enforces this invariant. In maxiphobic trees, the merge operation is implemented by comparing the roots of the two trees. The smaller root survives as the root of the merged tree. That leaves three sub-trees: the tree that lost in the comparison of the two roots, and the two child sub-trees of the winner. They are merged by first merging the two smaller trees, where the size of a tree is determined as the number of elements it contains, then attaching that merged tree along with the remaining tree as the children of the new root. The name maxiphobic, meaning "biggest avoiding", refers to the merge of the two smaller sub-trees.

a) Implement a Haskell module for maxiphobic heaps using following data type definition:

```
data Heap a = Empty | Node a Int (Heap a) (Heap a) deriving Show

-- number of elements
size :: Heap a -> Int
size Empty           = 0
size (Node _ sz _ _) = sz
```

b) (Difficult) Merge operation for maxiphobic heaps turns out to be logarithmic, so this is an efficient implementation of heaps. Prove that merge is indeed logarithmic.
Hint: Let T(N) be the number of merge functions invocations to merge two maxiphobic heaps when the total number of nodes in two merged heaps is N. Try to write a recurrence relation for T(N), noting that the biggest of the three sub-trees is avoided at each step. In order to determine total size of two sub-trees that will be merged (the two smaller ones), you should firstly determine how many elements does the avoided sub-tree (the biggest one) at least contain.

9. (Java) Implement maxiphobic heaps in Java.

10. (Haskell) Implement a module for dictionaries of keys and values by using an AVL tree. Each node in the tree should contain a key and associated value, and tree should be sorted according to key values.
Your implementation should define following functions:

- `empty :: Dictionary a b`. Returns an empty dictionary.
- `isEmpty :: Dictionary a b -> Bool`. Test for empty dictionary.
- `insert :: (Ord a) => a -> b -> Dictionary a b -> Dictionary a b`. Inserts a key and its value in a dictionary. If key was already present, old value is replaced by new one.

- `delete :: (Ord a) => a -> Dictionary a b -> Dictionary a b`. Removes node with provided key from dictionary.
- `valueOf :: (Ord a) => a -> Dictionary a b -> Maybe b`. returns (as a `Maybe`) value associated with key.
- `keys :: Dictionary a b -> [a]`. Returns sorted list with all keys in dictionary.
- `values :: Dictionary a b -> [b]`. Returns a list with all values in dictionary.
- `keysValues :: Dictionary a b -> [(a,b)]`. Returns list with all keys and values sorted according to keys values.
- `mapValues :: (Ord a) => (b -> c) -> Dictionary a b -> Dictionary a c`. Applies same function to all values in dictionary.

11. (Java) Implement a generic `Tuple2<A,B>` class in Java (in package `dataStructures.tuple`) to represent 2-components tuples, where `A` corresponds to type of first component and `B` to type of second one. In addition to constructor, include the following methods:

```
public A _1();
public B _2();
```

returning first and second components in the tuple and method `toString`, returning “Tuple2(x,y)”, where x and y stand for tuple components.

12. (Java) Implement dictionaries in Java. Implement all methods in exercise 10, except for `mapValues`. `keys`, `values` and `keysValues` should be implemented as iterables. Use class `Tuple2` of previous exercise for elements returned by `keysValues` iterator.

13. (Java) Implement bags by using an AVL tree. Keys should be elements in the bag and values should be corresponding number of occurrences. Compare efficiency of this implementation to the lineal one developed in exercises for unit 3, and do an experimental test with both implementations.

14. (Haskell) Arithmetic expressions can be represented by using a tree, so that operators are in internal nodes and values in leafs. Let us consider following data type for representing arithmetic expressions on trees:

```
data Expr = Value Integer
          | Add Expr Expr
          | Diff Expr Expr
          | Mult Expr Expr
          deriving Show
```

For instance, expression (1+2)*3 would be represented as:

```
e1 :: Expr
e1 = Mult (Add (Value 1) (Value 2)) (Value 3)
```

a) Define a function:

```
evaluate :: Expr -> Integer
```

to evaluate an expression.

b) If a tree corresponding to an arithmetic expression is traversed in post-order, RPN (reverse polish notation) representation of expression is obtained. Using this idea, define a function `toRPN` taking an `Expr` and returning a `String` with corresponding expression in RPN. For instance:

```
Main> toRPN e1
"1 2 + 3 *"
```

c) Now consider the following higher order function on expression:

```
foldExpr :: (Integer -> a) ->
            (a -> a -> a) ->
            (a -> a -> a) ->
            (a -> a -> a) ->
            Expr -> Integer
foldExpr f1 f2 f3 f4 e = fun e
  where
   fun (Value x)    = f1 x
   fun (Add e1 e2)  = f2 (fun e1) (fun e2)
   fun (Diff e1 e2) = f3 (fun e1) (fun e2)
   fun (Mult e1 e2) = f4 (fun e1) (fun e2)
```

This function replaces constructors `Value`, `Add`, `Diff` and `Mult` with functions `f1`,`f2`,`f3`, and `f4` respectively. Give an alternative definition of `evaluate` using `foldExpr` function:

```
evaluate' :: Expr -> Integer
evaluate’ e = foldExpr …
```

15. (Haskell) Mirror image of a tree.

a) Define function `mirrorB` taking a binary tree and returning its mirror image (tree as seen in a mirror)

b) Prove by induction that `mirrorB . mirrorB = id`

For this purpose, prove following base case and inductive step:

Base Case: `(mirrorB . mirrorB) EmptyB = EmptyB`

Inductive Step: If `(mirrorB . mirrorB) lt = lt` and `(mirrorB . mirrorB) rt = rt` then `(mirrorB . mirrorB) (NodeB x lt rt) = NodeB x lt rt`

16. (Haskell) Let a binary tree be symmetric if you can draw a vertical line through the root node and then the right sub-tree is the mirror image of the left sub-tree, taking into account only shapes of both trees, (i.e. ignoring specific values in the nodes of the trees). Write a function `isSymmetricB` to check whether a given binary tree is symmetric

17. (Haskell) Recall that a leaf is a node with no children, and any node of a tree that is not a leaf is an internal one.

a) Define function `leafsB` taking a binary tree and returning a list with values at leafs.

b) Define function `internalsB` taking a binary tree and returning a list with values at internal nodes.

c) Define function `leafs` and `internals` similar to previous ones but working on general trees.

18. (Haskell) Somebody represents binary trees of chars as strings of the following type:

```
a(b(d,e),c(,f(g,)))
```

Write functions to convert binary trees to strings and vice versa, for instance:

```
Main> stringToTreeB "x(y,a(,b))"
NodeB 'x' (NodeB 'y' EmptyB EmptyB) (NodeB 'a' EmptyB (NodeB 'b' EmptyB
EmptyB))
Main> treeToString $ NodeB 'x' (NodeB 'y' EmptyB EmptyB) (NodeB 'a'
EmptyB (NodeB 'b' EmptyB EmptyB))
"x(y,a(,b))"
```

19. (Haskell) Let the *internal path length* of a general tree be the total sum of the path lengths from the root to all other nodes of the tree. By this definition, the following tree has an internal path length of 9:

```
tree :: Tree Char
tree = Node 'a' [ Node 'f' [Node 'g' []]
                , Node 'c' []
                , Node 'b' [Node 'd' [], Node 'e' []]
                ]
```

Define a function `internalPL` to compute internal path length of its argument.

20. (Haskell) We'll define a root-to-leaf path to be a sequence of nodes in a tree starting with the root node and proceeding downward to a leaf (a node with no children). Write a function `allPaths` which returns a list of list representing all root-to-leaf paths for a general tree.

21. (Haskell) The following function has been presented in the slides to test whether a binary tree is a binary search tree (BST):

```
isBST :: (Ord a) => BST a -> Bool
isBST Empty          =  True
isBST (Node x lt rt) =  forAll (<x) lt && forAll (>x) rt
                         && isBST lt && isBST rt
 where
   forAll :: (a -> Bool) -> BST a -> Bool
   forAll p Empty          =  True
   forAll p (Node x lt rt) =  p x && forAll p lt && forAll p rt
```

Although this is a very clear definition, it is also a very inefficient one, as nodes in the tree are visited and compared many times. This problem can efficiently be solved by using an auxiliary function with two additional parameters:

```
inRange :: (Ord a) => a -> a -> BST a -> Bool
```

that traverses down the tree keeping track of the narrowing min and max interval values allowed in each sub-tree as it goes, looking at each node only once. The initial values for min and max should be maximum and minimum ones allowed for type corresponding to values in tree. In Haskell these extreme interval values can determined by using `minBound` and `maxBound` functions in class Bounded, so the initial call should be:

```
isBST' :: (Bounded a, Ord a) => BST a -> Bool
isBST' t = inRange minBound maxBound t
```

Define function `inRange` so that `isBST'` efficiently tests whether its argument is a BST.

22. (Haskell) Write a function `findTree` taking two lists corresponding to pre-order and in-order traversals of a binary tree and returning the corresponding tree. You may assume that there are no repeated elements in the tree. For instance:

```
Main> findTree [2,1,3,4,6] [3,1,4,2,6]
NodeB 2 (NodeB 1 (NodeB 3 EmptyB EmptyB) (NodeB 4 EmptyB EmptyB)) (NodeB
6 EmptyB EmptyB)
```

23. (Java) Solve previous Haskell exercises on trees, this time using Java.

24. Explain how to use a priority queue to implement a stack or a queue.

25. (Taken from Sedgewick and Wayne, 2010) Computational number theory. Write a program that prints out all integers of the form $a^3 + b^3$ where a and b are integers between 0 and N, in sorted order, without using excessive memory. That is, instead of computing an array of the N^2 sums and sorting them, build a minimum-oriented priority queue, initially containing $(0^3, 0, 0)$, $(1^3, 1, 0)$, $(2^3, 2, 0)$, . . . , $(N^3, N, 0)$. Then, while the priority queue is nonempty remove the smallest item $(i^3 + j^3, i, j)$, print it, and then, if $j < N$, insert the item $(i^3 + (j+1)^3, i, j+1)$. Use this program to find all distinct integers a, b, c, and d between 0 and 10^6 such that $a^3 + b^3 = c^3 + d^3$.

26. (Java) Write a program to count occurrences of words in a text file by using an AVL tree whose keys are words and whose values are number of occurrences. Your program will probably contain code similar to this one:

```
if (!avl.isElem(word)) avl.insert(word, 1);
else avl.insert(word, avl.search(word) + 1);
```

This is inefficient, as `isElem` calls `search(word)` and `search(word)` is called again immediately if the word is already included in the tree. So called *Software Caching* technique can be used to improve code like this one, where we save the location of the most recently searched key in a private instance variable, so that if the following search is on the same key, the location procedure does not need to be repeated. Add this optimization to AVL trees implementation.

27. (Java) Add efficient implementation for following operations to BSTs:

- `K floor(K key)`, returning largest key in BST less than or equal to key.
- `K ceiling(K key)`, returning smallest key in BST greater than or equal to key.
- `int rank(K key)`, returning number of keys in BST less than key.
- `K select(int i)` , returning i-th smallest key in BST (i=0 for smallest one).
- `void deleteMin()`, deleting smallest key in BST.
- `void deleteMax()`, deleting largest key in BST.
- `int size(Key lo, Key hi)`, returning number of keys in range lo to hi.

Note: in order to efficiently implement some of these operations, BST should be represented as augmented trees, so that each node also keeps total number of elements at tree rooted at that node.