TECHNICAL UNIVERSITY OF APPLIED SCIENCES ULM

Assignment #4

Student names and Group: *Petar Stoyanov & Eetu Hyvärinen - Group 08*

Course: *Algorithms and Data Structures* – Professor: *Prof. Schied*
Due date: *December 12th, 2019*

Exercise 4.2

The following interface is defined for priority queues:

```
public interface IPrioQueue<V extends IPrioritized> {
    void insert(V value);
    V extractMin();
    boolean isFull();
    boolean isEmpty();
}
```

Entries for the queue must implement the interface IPrioritized, i.e. be assigned an int-value as priority. A smaller number means a higher priority. If there are several entries with the same priority, then it is unspecified, which is removed first by extractMin().

```
public interface IPrioritized {
    int getPriority();
}
```

1. Program a class **HeapQueue<E>**, which efficiently implements a priority queue using a minimum heap. The maximum number of elements that can be stored is determined by the constructor.

   ```
   public class HeapQueue<V extends IPrioritized>
      implements IPrioQueue<V>{
       public HeapQueue(int maxSize){ //... }
       //...
   }
   ```

   Do not use the class **PriorityQueue** from the Java standard library! In Moodle, you can find test class **JuTestHeapQueue** for testing your implementation.
   Tip: You can reuse parts of Heapsort's program code with minor adjustments.

2. The program **PrioQueueMeasurement** determines the runtimes for problem sizes **n** = 100, 1 000,..., 1 000 000 for the following usage scenario:

   (*1*) First, **n** randomly selected elements are inserted into the queue.

   (*2*) Then 100 times the following is done: an element with randomly selected priority is inserted and an element is removed with **extractMin()**.

   (*3*) Finally, all **n** entries are removed from the queue with **extractMin()**.

Use it to measure your implementation. Give the measurement results and explain whether the theoretically expected runtime behavior can be recognizee from the measured values.

(*a*) **All code samples are submitted in the .java files along with the documentation**

(*b*) Final Measurements:

HeapQueue:

n	n x insert	100 x insert+extractMin	n x extractMin
100	0.36	0.42	0.01
1000	4.38	0.37	3.13
10000	138.60	4.70	59.73
100000	43216.47	116.17	3653.24

The expected runtime complexity is observed.

From these results we can expect the time it takes for n= 1 000 000 to be around 100 times longer than the time it took n = 100 000 as the runtime complexity is $O(n^2)$.

That would mean 4321600ms = 4321 seconds = 72.02 minutes = 1.2 hours

Exercise 4.3

1. Implement a priority queue with unbound capacity that implements the interface **IPrioQueue<E>** from exercise 4.2 as a linked list. In Moodle you will find a class template **exercise4_3_priolist.LinkedPrioQueue<E>** and a JUnit test class **JuTestLinkedPrioQueue**.
2. Analyze the runtime complexity of operations **insert(e)** and **extractMin()** in the average case.
3. Use class**LinkedPrioQueueMeasurement** to perform the same runtime measurements as in task 4.3. Can the expected runtime behavior be recognized?

(*a*) **All code samples are submitted in the .java files along with the documentation**

(*b*) extractMin() has a runtime complexity of O(1) as we only need to reassign the pointer to the first node./newline insert(e) has a runtime complexity of O(n) as we need to iterate through the whole list.

(*c*) Final Measurements:

LinkedPrioQueue:

n	n x insert	100 x insert+extractMin	n x extractMin
100	1.02	0.68	0.03
1000	7.71	0.74	0.05
10000	287.85	7.42	0.54
100000	98787.00	191.39	5.93

The expected runtime complexity is observed.

From these results we can expect the time it takes for n= 1 000 000 to be around 100 times longer than the time it took n = 100 000 as the runtime complexity is O(n^2).

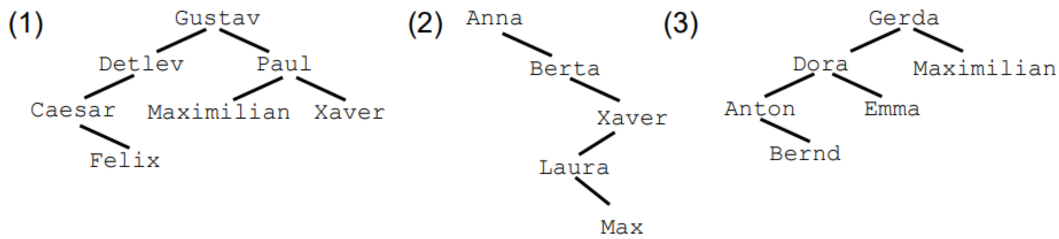
That would mean 9878700ms = 9878.7 seconds = 164.645 minutes = 2.744 hours

Runtime complexity for the Priority Queue using Linked list proved to be slower. One of the reasons being that in a linked list you need to take care of the pointer to the next node. In our case this was done behind the scenes but it drastically affected the code's performance

Exercise 4.5

Binary search trees should be used to store sets of strings. The usual lexicographic order is used for comparison.

1. Three trees are shown here:



   Which of the trees are binary search trees, which ones are not?
   Give a brief explanation.

2. Insert the strings **Gurke, Dattel, Ananas, Brot, Marmelade, Eis, Dosenmilch, Axt, Ente** and **Chili** into an initially empty search tree

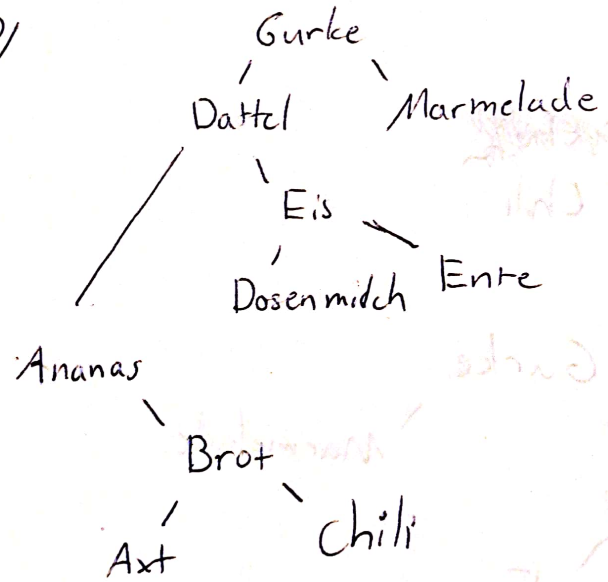
3. Now delete **Ananas, Datteln** and **Gurke** from the from the tree created in b) .

4.5 a) n.1 is not a binary search tree. Felix is on the right side of Caesar instead of being on the right side of Detler. F is not before D in the alphabet.

n. 2 is a binary search tree, but it is linear, inefficient

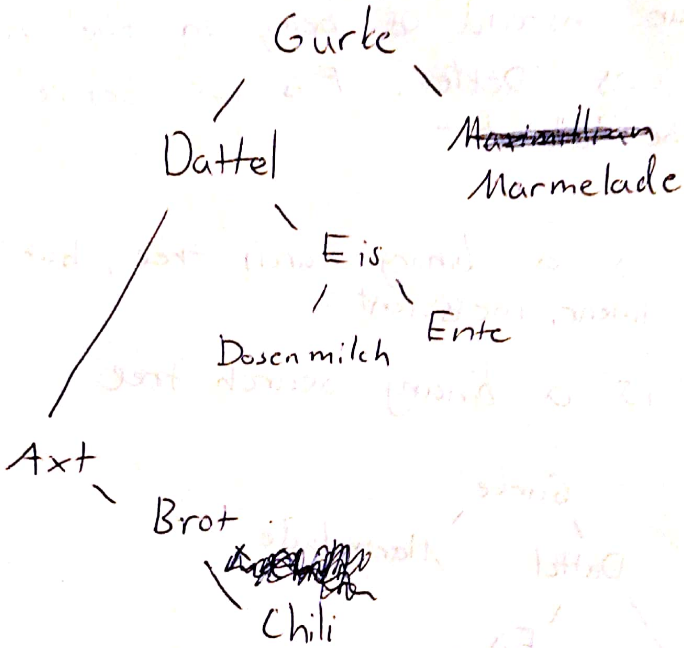
n. 3 is a binary search tree

b)

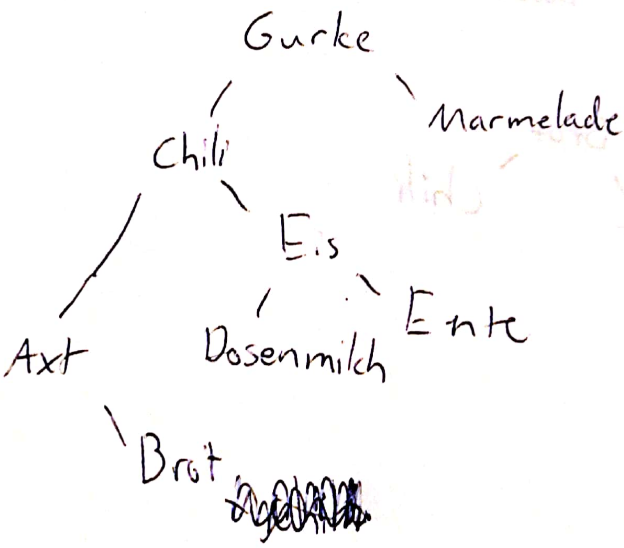


c) Remove Ananas, Dattel and Gurke:

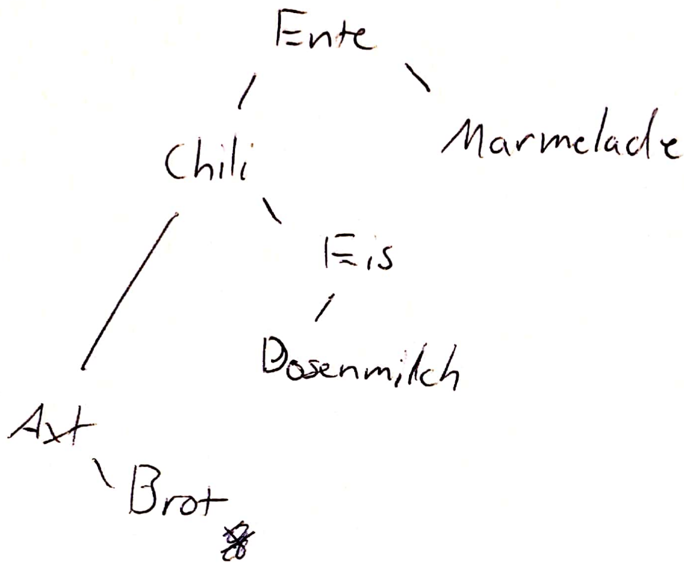
1. Ananas:



2. Dattel:



3. Gurke:



Exercise 4.6

(*a*) In Moodle you will find the classes **SearchTree** and **TreeNode** with the implementation of a binary search tree for values of type **double**. Enhance class **SearchTree** with the following methods:

```
public double sum()
    //Computes the sum of all values in the tree
public int countNegative()
    //Calculates the number of nodes with negative values
public int leaves()
    //Determines the number of leaves in the tree
public double removeMax()
    //Removes the node with the maximum value from the
       tree and returns its value.
    //Throws a RuntimeException if the tree is empty.
public ArrayList<Integer> toListDescending()
    //Returns the values stored in the tree sorted in
       descending order as ArrayList (package java. util)
    //(Tip: tree traversal)
public boolean equals(SearchTree other)
    //Checks whether the other tree contains exactly the
       same set of values, regardless of the structure of
       the tree. The method should be as efficient as
       possible.
```

(*b*) Which runtime complexity have **removeMax()** and **equals()** methods in the average and worst case, depending on the number n of elements in the tree? For **equals()** you may assume that both trees contain n values.

(*c*) Measure the runtime using the example program **SearchTreeMeasurement**

(a) of the combination**removeMax()** + **insert()** and

(b) of the **equals()** method

for randomly generated trees. Explain whether the expected runtime complexity is recognizable.

Answer.

(*a*) **All code samples are submitted in the .java files along with the documentation**

(*b*) removeMax() has a runtime complexity of O(logn) in both cases as we only look at the right side of the tree and the max node is always the most right node.

equals() has a runtime complexity from O(n logn) to O(n^2) depending on the sorting algorithm that is used.

(*c*) Final Measurements:

```
Runtime measurement insert()+removeMax():
n =         10: runtime per removeMax+insert    449.5 usec.
n =        100: runtime per removeMax+insert    150.5 usec.
n =       1000: runtime per removeMax+insert    326.9 usec.
n =      10000: runtime per removeMax+insert    215.8 usec.
n =     100000: runtime per removeMax+insert    717.9 usec.
n =    1000000: runtime per removeMax+insert    886.9 usec.

Runtime measurement equals():
n =       10:   true, runtime     0.70 msec.
n =      100:   true, runtime     0.50 msec.
n =     1000:   true, runtime     3.28 msec.
n =    10000:   true, runtime     7.43 msec.
n =   100000:   true, runtime    41.57 msec.
n =  1000000:   true, runtime   792.65 msec.
```

The expected runtime complexity is recognized.
Quick check for O(n logn) would be:

41ms - 7ms = 34ms

(44ms+34ms)*10 = 78*10 = 780ms expected runtime, observed runtime 898ms.

Exercise 4.9

(*a*) Starting from an empty AVL tree, insert the values

2, 3, 8, 10, 9, 5, 1, 11.

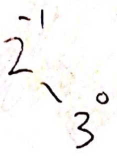
Specify the AVL tree after each insert operation and explain which rotations are necessary for restructuring

(*b*) Then delete the values 8 and 2 from the AVL tree one after the other. Think about how rotations can be used to restore the balance property.

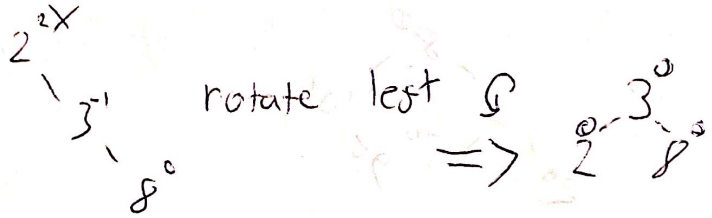
4.9 Insert 2:

2^{0} ok balance factor

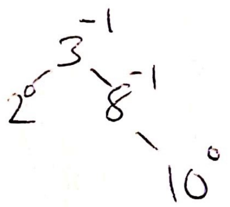
Insert 3:



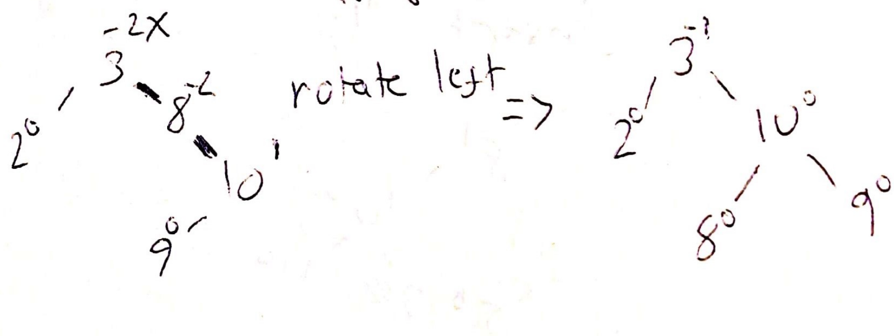
Insert 8



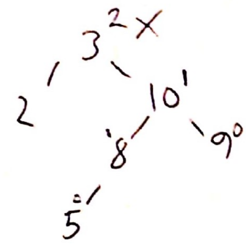
Insert 10



Insert 9

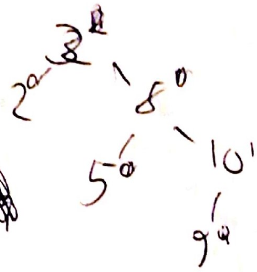


Insert 5:

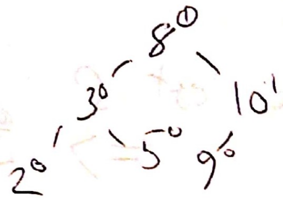


Double rotation

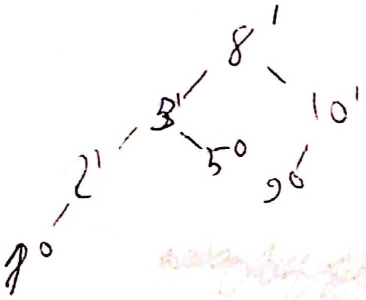
rotate right at 10



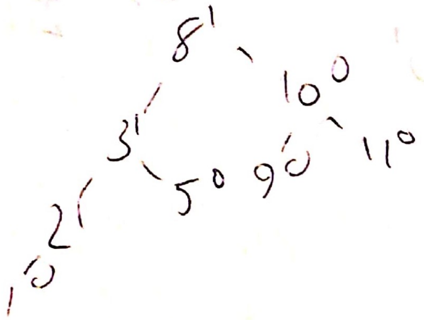
2nd rotate left at 3:



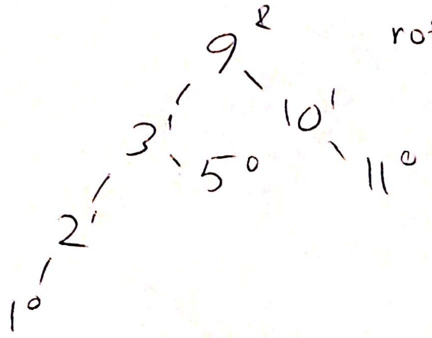
Insert 1:



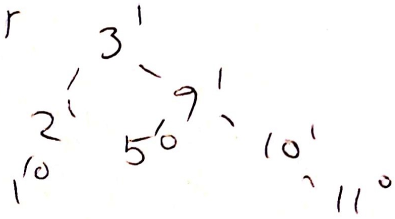
Insert 11:



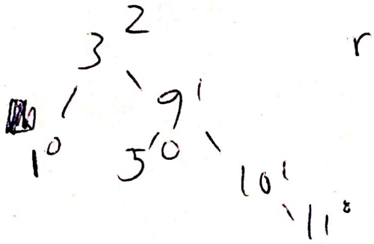
b) delete 8



rotate right =>



remove 2:



rotate left =>