Algorithms and Data Structures - Assignment 4

Graphs, heaps, sorting algorithms

Task A) Theory questions

1) An internal node of a binary tree has always two child nodes?

a. True
b. False
c. Depends on the height of the tree.
d. An internal node has no children.

Only complete, perfect and full binary trees.

2) We have a binary tree that is full at every level (this is known as a *perfect binary tree*). Which of the following formulas are true (h is height, L is number of leaves, n is number of nodes)?

a. $n = 2^{(h+1)} - 1$
b. $n = 2^h$
c. $L = 2^h$
d. $L = 2^{(h+1)} - 1$

3) In which of these three tree traversal of a binary tree is the root always at the first place?

a. pre-order
b. post-order
c. in-order
d. euler tour traversal

4) What is required for a binary tree structure?

a. It is necessary to know the children indices of every node.
b. It is necessary to keep track of the depth in every node.
c. It is necessary to have a root node.

If "children indices" is some sort of an identifier.

d. It is necessary to store the neighborhood information in every node.

5) A graph that contains an eulerian path can hold how many vertices with an odd degree at most?

a. none
b. 1
c. 2
d. any

Actually it depends on which category 0 falls in.
Normally it would be even, since it fits the definition of an even number.
If it would be odd then d) any would be correct.

6) Can a graph containing an euler path also have an euler circuit?

a. Yes, an euler path implies an euler circuit.
b. No, it either contains an euler path or an euler circuit.
c. Yes, but it does not imply it.
d. No, graph contains either both or none.

The graph could be not connected.

Task B) Euler Tour

The corresponding files for this task are in the *Task B* folder.

1) Add an edge

In the file *node.cpp* you have the skeleton of a method named void **add_edge(node* newNode)** which should add a new edge between the node which calls the method and the given node *newNode*.

2) Print Description

In the file *eulerpath.cpp* you have the skeleton of a method named void **print_graph(graph* newGraph)** which should print out a small overview of the nodes and edges in the given graph. Complete the function with your own implementation.

3) Euler Tour

We provide you a function skeleton named int **eulertour(graph* eulergraph)** which should return 1 if the given graph has an eulerian path and -1 if it hasn't. Make sure your code works with the provided graph.

Task C) Heap-datastructure

In this exercise you will learn to make your own Heap. You will also learn how to use STL-Pair, and will go over vectors and exceptions again. Once again we have a football team, Chelsea. Each player has a number and a name (a **std::string**), and the heap should index them by number. The basis for the heap is a vector and it's filled with the STL-Pairs. The respective files for this task are in the *Task C* folder.

1) Implementation

Implement all the functions declared in *Heap.h* with templates and STL-Pair in *Heap.cpp*.

Pay attention as well to the fact that the class definition is not given, only the methods.

Hint: The method definitions give you some hints how the data structure should be implemented.

2) Exception

Write your own *Exception* class and implement it in your methods to handle the empty heap (create also an *EmptyContainerException* class for this, which should inherit from *Exception*). The catch structure is already given in the *main.cpp* file.

Task D) Priority Queue based on a heap

In this task, you are asked to implement the functionality of a priority queue. This should be done by using heap functions, instead of classical arrays or node based priority queues. If you can't imagine how to implement a priority queue based on a heap: Remember that a from a heap you can always remove the smallest/largest element. So the heap preserves an order on the elements. The priorities in your priority queue can be managed with this heap approach. Your task is to fill in the missing method/function bodies.

Read the instructions on the linked sites (in the code) first. The corresponding files for this task are in the *Task D* folder.

Task E) Insertion Sort and Quicksort

In this exercise, we practice a couple of sorting algorithms. The corresponding files for this task are in the *Task E* folder.

1) Implementation of Insertion Sort

Implement an insertion sort algorithm. Complete the function **void Sort::insertion_sort(int arr[], int length)**.

2) Implementation of Quicksort

Implement a quicksort algorithm. Complete the function **void Sort::quicksort(int arr[], int left, int right)**. You can add helper functions if you wish.