Algorithms and Data Structures - Assignment 3

Lists, vectors, trees, maps

Task A) Theory questions

1) Which function is not part of the vector ADT?

a. push_back()
b. next()
c. front()
d. pop_back()

2) How many steps in big-O notation does it take to search an element in a binary tree?

a. $O(n^2)$
b. $O(\log n)$
c. $O(n)$
d. $O(2n \log n)$

3) How fast is an insertAfter() function on a list ADT?

a. $O(1)$
b. $O(n)$
c. $O(\log n)$
d. $O(n + 1)$

4) Searching an element in a doubly linked list is more efficient than a single linked list?

a. Yes, because you can start in the middle and search in both directions.
b. Yes, because you can start at head or tail.
c. No, the efficiency is the same.
d. No, there is difference between a single and a doubly linked list.

5) If you know one element of a doubly linked list, can you access all other elements?

a. Only if it is either the head or the tail.
b. No, because you can only access the elements in one direction.
c. Yes, because you can traverse in both directions.
d. You have to know the head and the tail to access all the elements.

6) Which statements are true?

a. The post-order traversal of a binary tree visits a parent element before a child element.
b. The pre-order traversal of a binary tree visits the parent element before the child element.
c. The post-order traversal of a binary tree visits the root element before a child element.
d. The post-order traversal of a binary tree visits leaf elements before parent elements.

7) What are the minimum and maximum heights that a binary tree with n nodes can have?

a. $\lfloor \log_2 n \rfloor$ and n.
b. $\lceil \log_2 n \rceil - 1$ and $n - 1$.
c. $\log_2 n$ and $2 \log_2 n$.
d. $\lfloor \log_2 n \rfloor$ and $n - 1$.

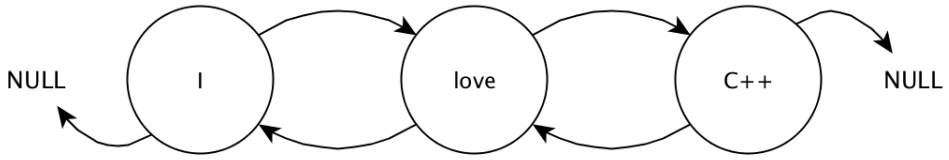
8) Which of these statements about the STL map is wrong?

a. It provides efficient access to key-value pairs.
b. You can access the elements by value.
c. Two distinct values may not have the same key.
d. You can access the elements by key.

Task B) Doubly linked list

A doubly linked list consists of any number of nodes. Each node is connected to both the previous and the next node (NULL if there isn't any). These connections are called links and implemented in C++ with pointers. It is very important that the first and the

last node point to NULL. This ensures that these nodes can be found in the list. You will find the relevant files in the folder *Task B*.



1) Implementation

You will find a code skeleton with a lot of empty methods to complete. The goal of the exercise is to implement a doubly linked list with similar functionalities to a C++ list. The method **pop_front()** for example has to correctly remove the first element of the doubly linked list.

Make sure that all links always point to the right node!

Task C) Vectors

In this exercise you will learn how to make your vector data structure and how to use templates for different datatypes. The vector should based on an array and has to contain all functions declared in the vectors.h file. It has to be also dynamic (handle the overflow by resizing) and using template. You will find the relevant files in the folder *Task C*.

1) Implementing Templates

Implement all the functions declared in *vectors.h* with templates in *vectors.cpp*.

2) Implementing a Float Array Vector

Write an example with a second ArrayVector and another datatype (for example float or double) to the *main.cpp*.

Task D) Node-based Binary Trees

In this exercise you will have to implement some functions of a simple (ordered) binary tree. In the framework we have provided the class KeyValuePair, which represents entities composed out of an integer key for searching and ordering, and of a value (**std::string**). We will use the tree for searching KeyValuePairs and for different tree traversals. The class you will have to add your code to is the class TreeNode, from which a binary tree is built. To get to know what the classes do and how they work, please refer to the headers and the main file. You will find the relevant files in the folder *Task D*.

Hint: All the functions you are programming will be called on the root node of the tree (see the main class). All your functions should use recursions.

1) Binary Tree Traversal

In the first task we ask you to fill in the code for the three traversal functions of the tree: **printPreOrder()**, **printPostOrder()** and **printInOrder()**. They should print the node values of the tree to the standard output, in the appropriate order. The header declarations of the functions are already provided. You just have to add the body in the source file.

Hint: Be careful, especially checking NULL pointers.

Hint: Please see the slide set *06_ Trees.pdf* in OLAT for detailed algorithmic instructions for the traversals.

2) Searching a Binary Tree

Now you have to program an important aspect of a binary tree: searching. The tree which is built in the main file is ordered. That means, that the left subtree of a node contains only smaller keys, and the right subtree contains only greater keys than the key of the node itself (assuming there are no duplicate keys for simplicity). This condition holds for every node in the tree.

Your task is to complete the function **searchKey(int key_)** which searches in the tree for a node with the key *key_*. If the node with the key is found, a pointer to this node should be returned. If the searched key doesn't exist in the tree, a pointer to the node where the search ended (that is, at one of the leaves) should be returned.

Hint: For algorithmic and theoretical help on binary tree searching, see *09_ BinarySearchTrees.pdf* in OLAT.

3) Inserting an Element in a Binary Tree

As a last subtask in this exercise, you are required to program the method **add(KeyValuePair* element_)** which is meant to create a new node out of the element_ and add that node on the right position in the tree. Remember that inserting in a binary ordered tree starts with searching for the key to insert. If it is already in there, you're finished. If not, the search for the key gives you the node that should be the parent of the new node.

Task E) std::Map

The aim of this task is to get an overview of the std::map structure. A bus is initially filled with 30 passengers. Each of them occupies one of the seats, which are named with strings. The names are already stored in the array *seats[]*, but passengers must be referenced by the seat's full name (you need to be able to find who is sitting in, for example, *"Row 3, seat B"*). To do this, you should use a *std::map<std::string,Passenger*>*. You will find the relevant files in the folder *Task D*.

1) Initialization of a Map

In the file *main.cpp* you have the skeleton of a method named **init()** which should initialize the map structure, i.e. assign all the 30 passengers to a seat. Complete the *for* loop with adding a new passenger to the map.

2) Implementing wait(int seconds)

We provide you a function skeleton named **wait(int seconds)** which should block the current process for a given amount of time. Complete the function so the process waits for the given seconds before resuming.

3) Implementing letPassengersOut()

We provide you a function skeleton named **letPassengersOut()** which should remove the Passengers in the map that have to leave the bus at the current station (i.e., delete them and set the pointer in the map to NULL). Use an iterator to traverse the whole map.

4) Implementing print()

We provide you a function skeleton named **print()** which should print out a little overview of the Passengers which are on the bus at the moment. Complete the function with your own implementation.

5) Implementing getNumberOfPassengers()

We provide you a function skeleton named **getNumberOfPassengers()** which should return the current number of passengers who are on the bus. Complete the function with your own implementation.