Exercise Sheet #7

Problem 1 (*Binary trees*) 10 Pts

In the lecture you saw an example code for writing a binary tree, where each node contains the pointers to its child nodes, allowing the user to traverse the tree downwards given a parent node. Change the code to create double linked trees, where each node contains a pointer to its parent as well, allowing you to traverse the whole tree starting from any point on it.

- Add a `parent` variable to the node class constructor. Change the function `generateTree` such that it initializes this variable for each node with its corresponding parent.
- Check that your implementation works by traversing the tree backwards: Starting with a node on the bottom layer, print out all its parents by accessing the `parent` variable repeatedly to climb up the tree. You can do that with a `while` loop if you add a stop condition for the root node.

Problem 2 (*File streams and searching*) 10 Pts

As seen in the lecture, the cursor inside a filestream can be moved with `seekg`/`seekp` to control where to read/write in the file. Given a text file with a list of numbers, you will use this possibility to implement a searching algorithm that searches a target number in the list and, if it is present, returns its position. You will implement a simple Binary Search Algorithm that requires ordered lists.

- Define an output filestream `ofstream myList`, so that the list will be written on a `numbers.dat` file.
- Generate a list of `n_numbers` random positive integers, with up to `n_digits` digits. Sort the list from low to high numbers using `std::sort()`.
- Write the list to the file. Later, you will have to move the cursor in the file to the positions where the numbers start. Therefore, to make things easier, fix the number of printed characters by using `myList.width(n_chars)`. Close the `ofstream` and define an `ifstream` to read from the generated file.
- Implement a function `fstream_binary_search` that takes a target integer and an `ifstream` as variables and returns the position in the list

of the target number if present, or -1 otherwise. Check that it works on a small 10 elements list and 3-digits numbers. Especially check that corner cases (such as the target number being at either end of the list) are correctly handled.

Problem 3 (*Overloading the stream operators << and >>*) 10 Pts

Following the example of the lecture implement a buffer class with a more extended functionality.

- The basic operations of this class should be the use of the operators << and >> to stream data into and out of a buffer, e.g.

```
int nNumber;
BufferClass mybuffer;
mybuffer << 3;
mybuffer << 4 << 5;
mybuffer >> nNumber;
```

  I.e. after streaming into nNumber the integer element 5 should not be in the buffer anymore.

- Design the class as a template class so that it can be used with different data types such as double or int, e.g.

```
BufferClass<int> int_buffer;
BufferClass<double> double_buffer;
```

- The size of the buffer should be specified via the constructor of the class, i.e.

```
BufferClass<int> int_buffer(10);
```

  for a buffer accepting 10 elements of type int.

- Implement methods to:
  - Reset the buffer to its initial state.
  - Obtain the current load of the buffer (number of saved elements).
  - Access information about the state of the buffer, specifically if it is empty or full.

- Make proper use of private and public classifications, i.e. if your class has an array to store the data type putting it into the private section forbids misusing it from the outside. The same should apply to all other data member elements of your class that should not be exposed directly to the user. For instance:

```
class BufferClass
{
public:
boolean checkIfEmpty()
{
return isEmpty;
}
private:
boolean isEmpty;
};
```

In this way the user is prevented of having direct access to isEmpty and avoid accidental modification of isEmpty. Instead the user can have access to isEmpty via checkIfEmpty() without being able to modify it.