Exercise Sheet #7

Deadline: 03.06.2024, 12:00h

Problem 1 (*Binary Trees*) (10 points)

A binary tree is a tree in which every node has at most two child nodes. In the lecture, the implementation of binary trees in `C++` has been discussed, where every node contained pointers to its child nodes, allowing the user to traverse the tree downwards starting from a given a parent node.
Your task in this exercise is, to change the code to create double linked trees, where each node additionally contains a pointer to its parent, allowing to traverse the whole tree starting from any node.

(a) 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.

(⑤ points)

(b) Check that your implementation works by traversing the tree backwards: Starting with a node from the bottom layer, print out all its parents by accessing the `parent` variable repeatedly to climb up the tree. Terminate, once you reach the root node.

(⑤ points)

Problem 2 (*File Streams and Searching*) (10 points)

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.

(a) Define an output file stream `ofstream myList`, so that the list will be written on a `numbers.dat` file.

(② points)

(b) Generate a list of N random positive integers, with up to N_d digits. Sort the list from low to high numbers using `std::sort()`.

(② points)

(c) 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.

(② points)

(d) 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. Try out your implementation with $N = 10$ and $N_d = 3$. Make sure to pay attention that corner cases (e.g. the target number is at either end of the list) are correctly handled.

(④ points)

Problem 3 (Advanced: *Overloading Stream Operators*) (10 points)

Following the example of the lecture, implement a buffer class with 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;
```

  In this example, after streaming into `nNumber`, the integer element `5` should not be in the buffer anymore.

- Design the class as a templated 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, e.g.

```
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.