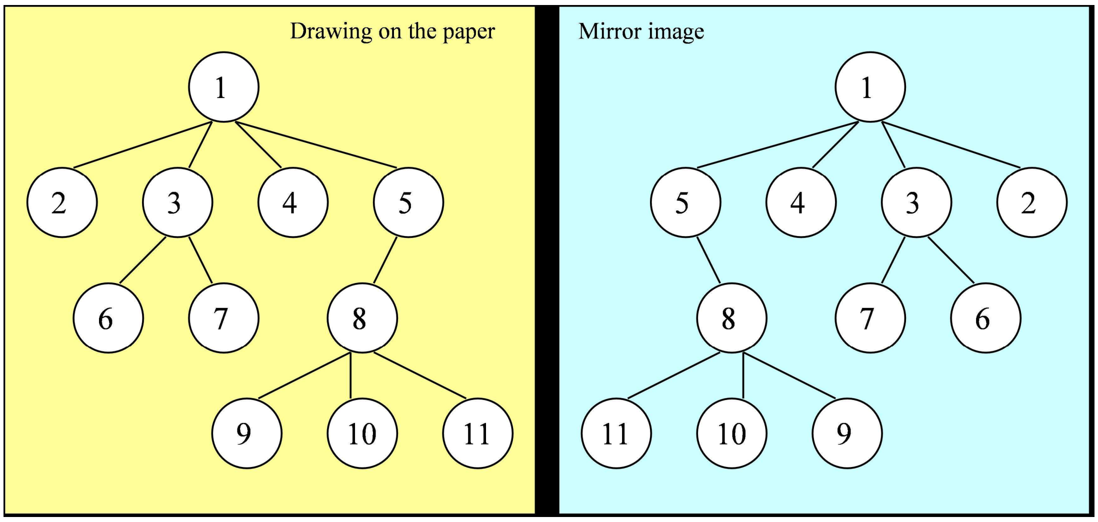
Programming task: Through a Looking Glass

Description

Alice has drawn a tree (in the IT sense), where nodes have unique numbers on the paper (look at *Drawing on the paper* in the picture). After a moment, Alice remembered her adventures as she went through the Looking-Glass. She immediately wanted to know, what this tree would look like in the mirror. Help Alice with this task and draw the mirror image of the tree (look at *Mirror image*).



A Tree and its Mirror Image

A computer understands a tree defined in the following format:
1. each line contains a record about a node that is not a leaf (internal node);
2. the first number of the record starts with a number of the internal node, followed by the numbers of its children in the left-to-right order.

If the order of these records is not strictly defined, then the tree shown in the Figure can have different representations. For example, it can be defined as:

```
1 2 3 4 5
3 6 7
5 8
8 9 10 11
```

or

```
5 8
3 6 7
8 9 10 11
1 2 3 4 5
```

The output tree should also be written in the same format. However, there is an additional condition that internal nodes are in *preorder* sequence, i.e. the parent's record always precedes the child record.

Your program should process trees written in any order in time no longer than $O(n\log n)$.

Alice is not keen on drawing trees. Consequently, a tree would not contain more than 10'000 nodes.

Input:

The input file contains zero or more records with the records of internal nodes of the tree in the following format:

```
Parent Child_1 Child_2 ... Child_n
...
0
```

- `Parent` shows the number of internal node of the tree [1..999'999'999]
- `Child_i` is the number of i-th child of *Parent* node (number [1..999'999'999])
- `0` - Input file always ends with the line containing only the number 0.

The input file is correct regarding the input data format and the given conditions.

Output:

The output file should contain a mirror image of the given tree. The tree should be written in the following way: internal nodes in *preorder* sequence.

Example 1 (input in *preorder*):

The content of input file `test01in.txt`:

```
1 2 3 4 5
3 6 7
5 8
8 9 10 11
0
```

The content of output file `test01expected.txt`:

```
1 5 4 3 2
5 8
8 11 10 9
3 7 6
0
```

Example 2 (input in *free order*):

The content of input file `test02in.txt`:

```
5 8
3 6 7
8 9 10 11
1 2 3 4 5
0
```

The content of output file `test02expected.txt`:

```
1 5 4 3 2
5 8
8 11 10 9
3 7 6
0
```

Implementation Details

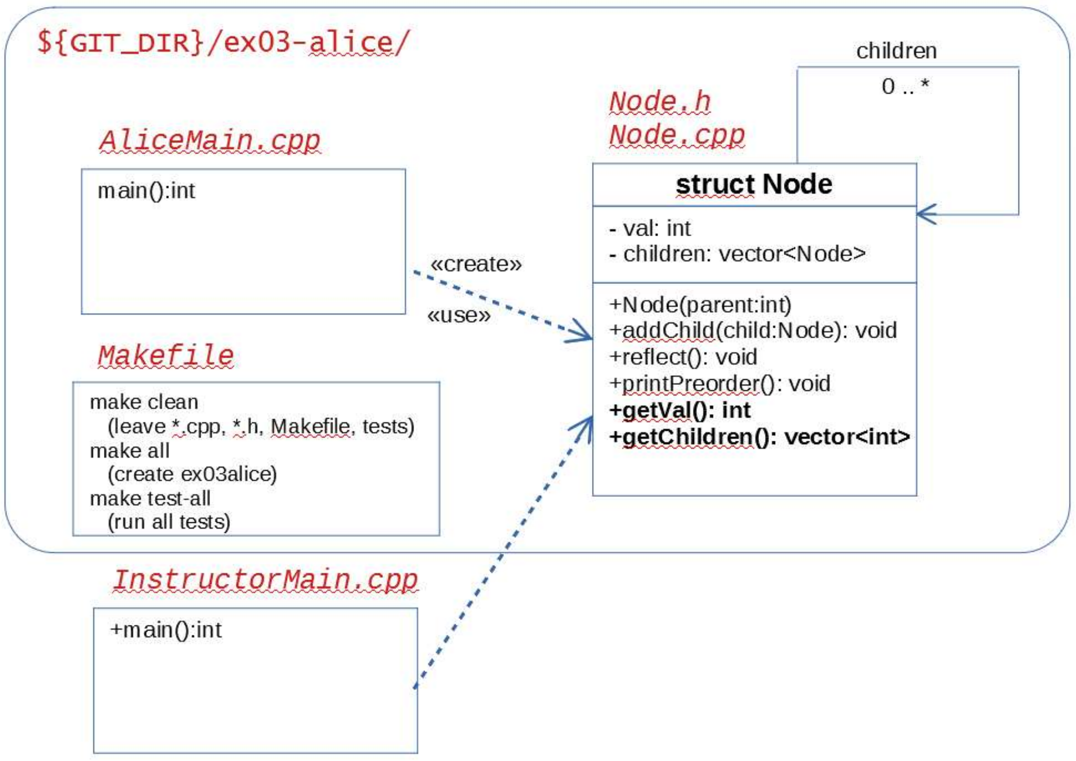
- **Use a standard subdirectory name:** Your solution should be located in subdirectory **ex03-alice** (all lowercase letters).
- **Use a standard Git branch name:** In the GitHub repository there should also be a branch of the same name:
  ex03-alice (this makes the grading process scalable – because we do not need to synchronize files in all the other directories).
- **Use the expected file names (they are case sensitive):** The source files that you need to rebuild and test your project should all be in the ex03-alice subdirectory - **AliceMain.cpp**, **Node.cpp** and **Node.h**.
  Your repository will also likely contain a **Makefile** (it can contain anything that is useful for your testing and debugging – because during the grading we will overwrite it with another one). It also would likely contain test files (INPUT files like test01in.txt) and expected results (like test01expected.txt).
- Do not keep useless files in the repository: Do not check in any executable files (**ex03alice** on Linux or **ex03alice.exe** on Windows etc.). Do not keep output files (test01out.txt etc.), object files and other stuff created by compiler. This would make your
- Your **Node** class or struct should be in the **ds_course** namespace (as every other class implemented before in this course). This avoids name collisions with similarly named classes that mean something else.
- Most of your grade (about 18 points out of 20 points) will reflect correct behavior of your code. The remaining grade will also test the Object Oriented design (grading process will provide the class InstructorMain.cpp – if your code does not implement the interfaces to link with it – you will lose about 2 points).

Grading script (for 90% of all the tests) will treat your code as a “black box”; it will call Linux executive file: **ex03alice** created by the Makefile in a way similar to this:

```
./ex03alice < in.txt > out.txt.
```

Object Oriented Design

The solution (that wants to get up to 100% of the grade, not just 90%) should implement the following UML Class Diagram:



“struct Node” or “class Node” should contain two private members: “val” (the number of the current node) and “children” (vector – i.e. a list of other Node elements). For leaf nodes this should be empty.

Moreover, there will be an alternative “main” method (InstructorMain.cpp) that will call the methods on your Node.cpp. (And include Node.h). You are not responsible for implementing this InstructorMain.cpp, but you should ensure that the following public methods are supported:

- Node(int vv) – a constructor assigning parameter “vv” to the private member “val”.
- addChild(child Node) – add one more child (to the right of all existing ones).

- reflect() – flip the existing Node (and all its successors – children, grandchildren, etc.) as a mirror image.
- printPreorder() – output the current node (and the whole subtree under it) in the preorder sequence.
- getVal() – return integer: the number of the current node.
- getChildren() – return vector of integrs: the numbers of the direct children.

Note 1: Please note that the current class design means that any node can navigate to its children, but a child node does not have a back-link to its parent. For some tasks you might need to move "upwards" in a tree (find the parent, grandparent, etc.), but for a simple DFS preorder traversal (and also – reflecting the tree as its mirror image) you only need down-links to the children.

Note 2: (getVal() and getChildren() might be irrelevant for producing the mirror images, but these "accessor methods" will make the task of evaluating in InstructorMain.cpp much easier.)