```cpp
#include <iostream>
#include <deque>
#include <string>
#include <map>
#include "Ex02.h"

// Though this wasn't part of the exam, I'm going to write program that works
// with a tree with repeated names printed in the correct order.

// I can simply overload the same functions with std::deque and make the
// appropriate changes inside the functions.
// For the printing function, the code will be exactly the same so a template
// function can be created (in the header file).


////////////////////////////////////////////////////////////////////////////


// An additional level of checking is necessary to avoid repeating names when
// returning from the left node to move to the right node (with set, repetitions
// are ignored so there's no problem).

using DQST = std::deque<std::string>;

void computeParentNodes(Node * n, DQST & parents) {
    if (n->left || n->right) {
        parents.push_back(n->name);
        if (n->left)
            computeParentNodes(n->left, parents);
        if (n->right)
            computeParentNodes(n->right, parents);
    }
}


////////////////////////////////////////////////////////////////////////////


// I believe the idea in the assignment was that when a sub-tree is printed
// names are printed level by level (see below). With a tree built like in
// this exercise is a bit challenging because the access point to a given
// node comes from one other node in the level above in the sub-tree and
// there is no easy way to scan nodes on the same level.
//
// To understand: in the folloring Alice is level 1; Bob and Carl are level
// 2, Emma, Daisy, and Bob are level 3, Joe is level 4, and so on.
//
//            Emma
//           /
//       Carl
//      /    \
// Alice       Bob
//      \          \
//       Bob        Joe
//           \
//            Daisy
//   1    2     3    4
//
```

```cpp
// names must be printed in level order so:
// Alice, Bob, Carl, Daisy, Bob, Emma, Joe

// If I want to print out the names on Lv. 3, I can only access these from 2
// different nodes in Lv. 2. This task is difficult to automate and to
// generalize and the deeper the sub-tree, the more complicated it becomes.
//
// The following solution may not be great but does the job.
// One variable to keep track of the level declared as static to keep its
// value in between recursive calls. This variable is used as a key for a
// static map and the values of this map is a std::deque.
// As the program moves up and down in the sub-tree, names are pushed in the
// deque values indexed by 'level' and any level that is addressed for the
// first time is automatically created.
// The top level is 1.
//
// The final condition (level == 1) is reached only when the processing of the
// top levels is over and the available sub-tree from that node has been
// completely scanned. At that point, the function goes through the map and
// pushes names into 'members' in the correct level order.
// After this the map must be cleared or in subsequent calls the static map
// will still have the same content as previous computation.


using DQST = std::deque<std::string>;

void computeMembersOfSubTree(Node * n, DQST & members) {
    static int level = 1;
    static std::map<int , DQST> names;

    names[level].push_back(n->name);

    if (n->left) {
        level++;                                        // Move DOWN the tree
        computeMembersOfSubTree(n->left, members);      //
        level--;                                        // Move UP
    }
    if (n->right) {
        level++;                                        // Move DOWN the tree
        computeMembersOfSubTree(n->right, members);     //
        level--;                                        // Move UP
    }

    // Transfer names from the map to 'members'
    if (level == 1) {
        for (auto& gen : names) {
            for (auto n : gen.second) {
                members.push_back(n);
            }
        }
        names.clear();
    }
}
```