EADS laboratory task 3

Author: Sławomir Batruch

Version: 2

Task type: AVL data structure design

The task for this laboratory was to design a Dictionary class, which is actually a self-balancing tree, also known as AVL. It has the same rules as the BST, but it balances itself so that any of the subtrees is not too long.

Additional task was to design the following methods. Unfortunately, I did not design them, but here are the declarations of them:

```
//---------------------------------
Dictionary <string, int>& counter (const string& fileName){
//---------------------------------
//
// counter (a Dictionary) of words from a text file (count word occurrences);
// all words encountered in a text file together with the word occurrence count
// ordered lexically
...
}
//---------------------------------
Ring <int, string>& listing (const Dictionary <string, int>&){
//---------------------------------
//
// listing (a Ring) of words from a Dictionary ordered by word occurrence count;
// moreover words with the same counter value ordered lexically within the listing
//
...
}
```

Now, here are the methods and data structure design I did. I added needed comments to these declarations. Additional explanation and information (which is a part of documentation, but internally in the file) can be found in the implementation part of the methods:

```
template<typename Key, typename Info>
struct Node {
    Key key;
    Info info;
    short int balance; // node balance factor (AVL)
    Node *left;
    Node *right;
};

template<typename Key, typename Info>
class Dict {
    public:
        Dict() { // Classic empty constructor
            this->root = nullptr;
        }

        ~Dict() { // Clears the memory by destroying every node
            destroy(root);
        }

        Dict(const Dict<Key, Info>& srcTree); // Copy constructor
        const Dict& operator=(const Dict& srcTree); // Operator=, similar to c
opy constructor. Refer to implementation
        // if bool verbose is true, then balance factors are also written out
        void inorderTraverse(bool verbose) const;
        void preorderTraverse(bool verbose) const;
        void postorderTraverse(bool verbose) const;

        unsigned int treeHeight() const; // Gives the height of the tree (long
est subtree)
        unsigned int treeNodeCount() const; // Gives the amount of nodes of th
e tree

        void clearTree(); // Clears the tree i.e. deletes all the nodes and se
ts them to nullptr

        void insert(const Key newKey, const Info newInfo); // public insert fu
nction

    private: // These methods are called by their respective public methods.
    // some other necessary methods might be shown here as well

        void copyTree(Node<Key, Info>*& thisTreeRoot, Node<Key, Info>* srcTree
Root); // used in deep copy ctor
        void destroy(Node<Key, Info>*& arg); // Used in clearTree

        void inorder(Node<Key, Info>* arg, bool verbose) const; // Used in ino
rderTraverse
```

```
        void preorder(Node<Key, Info>* arg, bool verbose) const; // Used in pr
eorderTraverse
        void postorder(Node<Key, Info>* arg, bool verbose) const; // Used in p
ostorderTraverse

        unsigned int height(Node<Key, Info>* arg) const; // used in treeHeight

        unsigned int nodeCount(Node<Key, Info>* arg) const; // Used in treeNod
eCount

        // AVL METHODS:
        void rotateLeft(Node<Key, Info>*& arg); // does a left rotation
        void rotateRight(Node<Key, Info>*& arg); // does a right rotation

        void balanceFromLeft(Node<Key, Info>*& arg); // called when we need to
 move some node to the right
        void balanceFromRight(Node<Key, Info>*& arg); // called when we need t
o move some node to the left subtree

        void AVLinsert(Node<Key, Info>*& arg, Node<Key, Info>* newNode, bool&
taller); // internal insert into AVL method
        // taller is a "pass by reference" bool variable
        // It lets know if the insertion made the tree higher (because not alw
ays is the tree taller)
        template<typename typeA, typename typeB>
        int max(const typeA a, const typeB b) const { // Supportive function.
Returns the higher argument
            return (a >= b) ? a : b;
        }

        Node<Key, Info>* root; // The root of the tree
};
```

Note that default constructor, destructor and max function implementation is included in the class, as it can be seen from the pasted code snippet. For more extensive information please refer to the header file.

The way the class is done (general overview) is that public methods, which are available to the users, call their respective private methods, that do any bigger modifications or work to the data structure. That allows the user not to tamper by accident or intentionally with the data structure's variables.

Testing

The testing of the class was done by calling all the public methods that could be tested, that is:

- All the traversal methods
- Node amount and tree height values
- Copy constructor and operator=
- Node insertion

The whole process is semi-automatic as it required the user to look for themselves into the standard output and compare the outputted values to the ones, that should be – theoretically – present in the nodes of the AVL tree.

Compilation

The compiling and launching process is a 3 step classic procedure.

1. Compile the header `g++ <header_name>.hpp`
2. Compile the testing source code with output executable name as argument after `-o` arg.
   `g++ <source_file_name>.cpp -o <executable_name>.exe`
3. Launch the executable. Depends on the terminal, but example command is
   `./<executable_name>.exe`